Parallel Computer Architecture

A Hardware / Software Approach

David Culler
University of California, Berkeley

Jaswinder Pal Singh
Princeton University

with Anoop Gupta
Stanford University

Preface

Morgan Kaufmann is pleased to present material from a preliminary draft of Parallel Computer Architecture; the material is (c) Copyright 1997 Morgan Kaufmann Publishers. This material may not be used or distributed for any commercial purpose without the express written consent of Morgan Kaufmann Publishers. Please note that this material is a draft of forthcoming publication, and as such neither Morgan Kaufmann nor the authors can be held liable for changes or alterations in the final edition.

Motivation for the Book

Parallel computing is a critical component of the computing technology of the 90s, and it is likely to have as much impact over the next twenty years as microprocessors have had over the past twenty. Indeed, the two technologies are closely linked, as the evolution of highly integrated microprocessors and memory chips is making multiprocessor systems increasingly attractive. Already multiprocessors represent the high performance end of almost every segment of the computing market, from the fastest supercomputers, to departmental compute servers, to the individual desktop. In the past, computer vendors employed a range of technologies to provide increasing performance across their product line. Today, the same state-of-the-art microprocessor is used throughout. To obtain a significant range of performance, the simplest approach is to increase the number of processors, and the economies of scale makes this extremely attractive. Very soon, several processors will fit on a single chip.

Although parallel computing has a long and rich academic history, the close coupling with commodity technology has fundamentally changed the discipline. The emphasis on radical architectures and exotic technology has given way to quantitative analysis and careful engineering trade-offs. Our goal in writing this book is to equip designers of the emerging class of multiprocessor systems, from modestly parallel personal computers to massively parallel supercomputers, with an understanding of the fundamental architectural issues and the available techniques for addressing design trade-offs. At the same time, we hope to provide designers of software systems for these machines with an understanding of the likely directions of architectural evolution and the forces that will determine the specific path that hardware designs will follow.

The most exciting recent development in parallel computer architecture is the convergence of traditionally disparate approaches, namely shared-memory, message-passing, SIMD, and dataflow, on a common machine structure. This is driven partly by common technological and economic forces, and partly by a better understanding of parallel software. This convergence allows us to focus on the overriding architectural issues and to develop a common framework in which to understand and evaluate architectural trade-offs. Moreover, parallel software has matured to the point where the popular parallel programming models are available on a wide range of machines and meaningful benchmarks exists. This maturing of the field makes it possible to undertake a quantitative, as well as qualitative study of hardware/software interactions. In fact, it demands such an approach. The book follows a set of issues that are critical to all parallel architectures – communication latency, communication bandwidth, and coordination of cooperative work - across the full range of modern designs. It describes the set of techniques available in hardware and in software to address each issue and explores how the various techniques interact. Case studies provide a concrete illustration of the general principles and demonstrate specific interactions between mechanisms.

Our final motivation comes from the current lack of an adequate text book for our own courses at Stanford, Berkeley, and Princeton. Many existing text books cover the material in a cursory fashion, summarizing various architectures and research results, but not analyzing them in depth. Others focus on specific projects, but fail to recognize the principles that carry over to alternative approaches. The research reports in the area provide sizable body of empirical data, but it has not yet been distilled into a coherent picture. By focusing on the salient issues in the context of the technological convergence, rather than the rich and varied history that brought us to this point, we hope to provide a deeper and more coherent understanding of the field.

Intended Audience

We believe the subject matter of this book is core material and should be relevant to graduate students and practicing engineers in the fields of computer architecture, systems software, and applications. The relevance for computer architects is obvious, given the growing importance of multiprocessors. Chip designers must understand what constitutes a viable building block for multiprocessor systems, while computer system designers must understand how best to utilize modern microprocessor and memory technology in building multiprocessors.

Systems software, including operating systems, compilers, programming languages, run-time systems, performance debugging tools, will need to address new issues and will provide new opportunities in parallel computers. Thus, an understanding of the evolution and the forces guiding that evolution is critical. Researchers in compilers and programming languages have

addressed aspects of parallel computing for some time. However, the new convergence with commodity technology suggests that these aspects may need to be reexamined and perhaps addressed in very general terms. The traditional boundaries between hardware, operating system, and user program are also shifting in the context of parallel computing, where communication, scheduling, sharing, and resource management are intrinsic to the program.

Applications areas, such as computer graphics and multimedia, scientific computing, computer aided design, decision support and transaction processing, are all likely to see a tremendous transformation as a result of the vast computing power available at low cost through parallel computing. However, developing parallel applications that are robust and provide good speed-up across current and future multiprocessors is a challenging task, and requires a deep understanding of forces driving parallel computers. The book seeks to provide this understanding, but also to stimulate the exchange between the applications fields and computer architecture, so that better architectures can be designed --- those that make the programming task easier and performance more robust.

Organization of the Book

The book is organized into twelve chapters. Chapter 1 begins with the motivation why parallel architectures are inevitable based on technology, architecture, and applications trends. It then briefly introduces the diverse multiprocessor architectures we find today (shared-memory, message-passing, data parallel, dataflow, and systolic), and it shows how the technology and architectural trends tell a strong story of convergence in the field. The convergence does not mean the end to innovation, but on the contrary, it implies that we will now see a time of rapid progress in the field, as designers start talking *to* each other rather than *past* each other. Given this convergence, the last portion of the chapter introduces the fundamaental design issues for multiprocessors: naming, synchronization, latency, and bandwidth. These four issues form an underlying theme throughout the rest of this book. The chapter ends with a historical perspective, poviding a glimpse into the diverse and rich history of the field.

Chapter 2 provides a brief introduction to the process of parallel programming and what are the basic components of popular programming models. It is intended to ensure that the reader has a clear understanding of hardware/software trade-offs, as well as what aspects of performance can be addressed through architectural means and what aspects much be addressed either by the compiler or the programmer in providing to the hardware a well designed parallel program. The analogy in sequential computing is that architecture cannot transform an $O(n^2)$ algorithm into an $O(n\log n)$ algorithm, but it can improve the average access time for common memory reference patterns. The brief discussion of programming issues is not likely to turn you into an expert parallel programmer, if you are not already, but it will familiarize you with the issues and what programs look like in various programming models. Chapter 3 outlines the basic techniques used in programming for performance and presents a collection of application case studies, that serve as a basis for quantitative evaluation of design trade-offs throughout the book.

Chapter 4 takes up the challenging task of performing a solid empirical evaluation of design trade-offs. Architectural evaluation is difficult even for modern uni-processors where we typically look at variations in pipeline design or memory system design against a fixed set of programs. In parallel architecture we have many more degrees of freedom to explore, the interactions between aspects of the design are more profound, the interactions between hardware

and software are more significant and of a wider scope. In general, we are looking at performance as the machine and the program scale. There is no way to scale one without the other example. Chapter 3 discusses how scaling interacts with various architectural parameters and presents a set of benchmarks that are used throughout the later chapters.

Chapters 5 and 6 provide a complete understanding of the bus-based multiprocessors, SMPs, that form the bread-and-butter of modern commercial machines beyond the desktop, and even to some extent on the desktop. Chapter 5 presents the logical design of "snooping" bus protocols which ensure that automatically replicated data is conherent across multiple caches. This chapter provides an important discussion of memory consistency models, which allows us to come to terms with what shared memory really means to algorithm designers. It discusses the spectrum of design options and how machines are optimized against typical reference patterns occuring in user programs and in the operating system. Given this conceptual understanding of SMPs, it reflects on implications for parallel programming.

Chapter 6 examines the physical design of bus-based multiprocessors. Itdigs down into the engineering issues that arise in supporting modern microprocessors with multilevel caches on modern busses, which are highly pipelined. Although some of this material is contained in more casual treatments of multiprocessor architecture, the presentation here provides a very complete understanding of the design issues in this regime. It is especially important because these small-scale designs form a building block for large-scale designs and because many of the concepts will reappear later in the book on a larger scale with a broader set of concerns.

Chapter 7 presents the hardware organization and architecture of a range of machines that are scalable to large or very large configurations. The key organizational concept is that of a network transaction, analogous to the bus transaction, which is the fundamental primitive for the designs in Chapters 5 and 6. However, in large scale machines the global information and global arbitration of small-scale designs is lost. Also, a large number of transactions can be outstanding. We show how conventional programming models are realized in terms of network transactions and then study a spectrum of important design points, organized according to the level of direct hardware interpretation of the network transaction, including detailed case studies of important commercial machines.

Chapter 8 puts the results of the previous chapters together to demonstrate how to realize a global shared physical address space with automatic replication on a large scale. It provides a complete treatment of directory based cache coherence protocols and hardware design alternatives.

Chapter 9 examines a spectrum of alternatives that push the boundaries of possible hardware/software trade-offs to obtain higher performance, to reduce hardware complexity, or both. It looks at relaxed memory consistency models, cache-only memory architectures, and software based cache coherency. This material is currently in the transitional phase from academic research to commercial product at the time of writing.

Chapter 10 addresses the design of scable high-performance communication networks, which underlies all the large scale machines discussed in previous chapters, but was deferred in order to complete our understanding of the processor, memory system, and network interface design which drive these networks. The chapter builds a general framework for understanding where hardware costs, transfer delays, and bandwidth restrictions arise in networks. We then look at a variety of trade-offs in routing techniques, switch design, and interconnection topology with

respect to these cost-performance metrics. These trade-offs are made concrete through case studies of recent designs.

Given the foundation established by the first ten chapters, Chapter 11 examines a set of cross-cutting issues involved in tolerating significant communication delays without impeding performance. The techniques essentially exploit two basic capabilities: overlapping communication with useful computation and pipelinng the communication of a volume of data. The simplest of these are essentially bulk transfers, which pipeline the movement of a large regular sequence of data items and often can be off-loaded from the processor. The other techniques attempt to hide the latency incured in collections of individual loads and stores. Write latencies are hidden by exploiting weak consistency models, which recognize that ordering is convey by only a small set of the accesses to shared memory in a program. Read latencies are hidden by implicit or explicit prefetching of data, or by look-ahead techniques in modern dynamically scheduled processors. The chapter provides a thorough examination of these alternatives, the impact on compilation techniques, and quantitative evaluation of the effectiveness. Finally, Chapter 12 examines the trends in technology, architecture, software systems and applications that are likely to shape evolution of the field.

We believe parallel computer architecture is an exciting core field whose importance is growing; it has reached a point of maturity that a textbook makes sense. From a rich diversity of ideas and approaches, there is now a dramatic convergence happening in the field. It is time to go beyond surveying the machine landscape, to an understanding of fundamental design principles. We have intimately participated in the convergence of the field; this text arises from this experience of ours and we hope it conveys some of the excitement that we feel for this dynamic and growing area.

CHAPTER 1   Introduction

1.1 Introduction

We have enjoyed an explosive growth in performance and capability of computer systems for over a decade. The theme of this dramatic success story is the advance of the underlying VLSI technology, which allows larger and larger numbers of components to fit on a chip and clock rates to increase. The plot is one of computer architecture, which translates the raw potential of the technology into greater performance and expanded capability of the computer system. The leading character is parallelism. A larger volume of resources means that more operations can be done at once, in parallel. Parallel computer architecture is about organizing these resources so that they work well together. Computers of all types have harnessed parallelism more and more effectively to gain performance from the raw technology, and the level at which parallelism is exlploited continues to rise. The other key character is storage. The data that is operated on at an ever faster rate must be held somewhere in the machine. Thus, the story of parallel processing is deeply intertwined with data locality and communication. The computer architect must sort out

these changing relationships to design the various levels of a computer system so as to maximize performance and programmability within the limits imposed by technology and cost at any particular time.

Parallelism is a fascinating perspective from which to understand computer architecture, because it applies at all levels of design, it interacts with essentially all other architectural concepts, and it presents a unique dependence on the underlying technology. In particular, the basic issues of locality, bandwidth, latency, and synchronization arise at many levels of the design of parallel computer systems. The trade-offs must be resolved in the context of real application workloads.

Parallel computer architecture, like any other aspect of design, involves elements of form and function. These elements are captured nicely in the following definition[AGo89].

> A *parallel computer* is a collection of processing elements that cooperate and communicate to solve large problems fast.

However, this simple definition raises many questions. How large a collection are we talking about? How powerful are the individual processing elements and can the number be increased in a straight-forward manner? How do they cooperate and communicate? How are data transmitted between processors, what sort of interconnection is provided, and what operations are available to sequence the actions carried out on different processors? What are the primitive abstractions that the hardware and software provide to the programmer? And finally, how does it all translate into performance? As we begin to answer these questions, we will see that small, moderate, and very large collections of processing elements each have important roles to fill in modern computing. Thus, it is important to understand parallel machine design across the scale, from the small to the very large. There are design issues that apply throughout the scale of parallelism, and others that are most germane to a particular regime, such as within a chip, within a box, or on a very large machine. It is safe to say that parallel machines occupy a rich and diverse design space. This diversity makes the area exciting, but also means that it is important that we develop a clear framework in which to understand the many design alternatives.

Parallel architecture is itself a rapidly changing area. Historically, parallel machines have demonstrated innovative organizational structures, often tied to particular programing models, as architects sought to obtain the ultimate in performance out of a given technology. In many cases, radical organizations were justified on the grounds that advances in the base technology would eventually run out of steam. These dire predictions appear to have been overstated, as logic densities and switching speeds have continued to improve and more modest parallelism has been employed at lower levels to sustain continued improvement in processor performance. Nonetheless, application demand for computational performance continues to outpace what individual processors can deliver, and multiprocessor systems occupy an increasingly important place in mainstream computing. What has changed is the novelty of these parallel architectures. Even large-scale parallel machines today are built out of the same basic components as workstations and personal computers. They are subject to the same engineering principles and cost-performance trade-offs. Moreover, to yield the utmost in performance, a parallel machine must extract the full performance potential of its individual components. Thus, an understanding of modern parallel architectures must include an in-depth treatment of engineering trade-offs, not just a descriptive taxonomy of possible machine structures.

Parallel architectures will play an increasingly central role in information processing. This view is based not so much on the assumption that individual processor performance will soon reach a

plateau, but rather on the estimation that the next level of system design, the multiprocessor level, will become increasingly attractive with increases in chip density. *The goal of this book is to articulate the principles of computer design at the multiprocessor level.* We examine the design issues present for each of the system components – memory systems, processors, and networks – and the relationships between these components. A key aspect is understanding the division of responsibilities between hardware and software in evolving parallel machines. Understanding this division, requires familiarity with the requirements that parallel programs place on the machine and the interaction of machine design and the practice of parallel programming.

The process of learning computer architecture is frequently likened to peeling an onion, and this analogy is even more appropriate for parallel computer architecture. At each level of understanding we find a complete whole with many interacting facets, including the structure of the machine, the abstractions it presents, the technology it rests upon, the software that exercises it, and the models that describe its performance. However, if we dig deeper into any of these facets we discover another whole layer of design and a new set of interactions. The wholistic, many level nature of parallel computer architecture makes the field challenging to learn and challenging to present. Something of the layer by layer understanding is unavoidable.

This introductory chapter presents the 'outer skin' of parallel computer architecture. It first outlines the reasons why we expect parallel machine design to become pervasive from desktop machines to supercomputers. We look at the technological, architectural, and economic trends that have led to the current state of computer architecture and that provide the basis for anticipating future parallel architectures. Section 1.2 focuses on the forces that have brought about the dramatic rate of processor performance advance and the restructuring of the entire computing industry around commodity microprocessors. These forces include the insatiable application demand for computing power, the continued improvements in the density and level of integration in VLSI chips, and the utilization of parallelism at higher and higher levels of the architecture.

We then take a quick look at the spectrum of important architectural styles which give the field such a rich history and contribute to the modern understanding of parallel machines. Within this diversity of design, a common set of design principles and trade-offs arise, driven by the same advances in the underlying technology. These forces are rapidly leading to a convergence in the field, which forms the emphasis of this book. Section 1.3 surveys traditional parallel machines, including *shared-memory, message-passing, single-instruction-multiple-data, systolic arrays*, and *dataflow*, and illustrates the different ways that they address common architectural issues. The discussion illustrates the dependence of parallel architecture on the underlying technology and, more importantly, demonstrates the convergence that has come about with the dominance of microprocessors.

Building on this convergence, in Section 1.4 we examine the fundamental design issues that cut across parallel machines: what can be named at the machine level as a basis for communication and coordination, what is the latency or time required to perform these operations, and what is the bandwidth or overall rate at which they can be performed. This shift from conceptual structure to performance components provides a framework for quantitative, rather than merely qualitative, study of parallel computer architecture.

With this initial, broad understanding of parallel computer architecture in place, the following chapters dig deeper into its technical substance. Chapter 2 delves into the structure and requirements of parallel programs to provide a basis for understanding the interaction between parallel architecture and applications. Chapter 3 builds a framework for evaluating design decisions in

terms of application requirements and performance measurements. Chapter 4 is a complete study of parallel computer architecture at the limited scale that is widely employed in commercial multiprocessors – a few processors to a few tens of processors. The concepts and structures introduced at this scale form the building blocks for more aggressive large scale designs presented over the next five chapters.

1.2 Why Parallel Architecture

Computer architecture, technology, and applications evolve together and have very strong interactions. Parallel computer architecture is no exception. A new dimension is added to the design space – the number of processors – and the design is even more strongly driven by the demand for performance at acceptable cost. Whatever the performance of a single processor at a given time, higher performance can, in principle, be achieved by utilizing many such processors. How much additional performance is gained and at what additional cost depends on a number of factors, which we will explore throughout the book.

To better understand this interaction, let us consider the performance characteristics of the processor building blocks. Figure 1-1[1] illustrates the growth in processor performance over time for several classes of computers[HeJo91]. The dashed lines represent a naive extrapolation of the trends. Although one should be careful in drawing sharp quantitative conclusions from such limited data, the figure suggests several valuable observations, discussed below.

First, the performance of the highly integrated, single-chip CMOS microprocessor is steadily increasing and is surpassing the larger, more expensive alternatives. Microprocessor performance has been improving at a rate of more than 50% per year. The advantages of using small, inexpensive, low power, mass produced processors as the building blocks for computer systems with many processors are intuitively clear. However, until recently the performance of the processor best suited to parallel architecture was far behind that of the fastest single processor system. This is no longer so. Although parallel machines have been built at various scales since the earliest days of computing, the approach is more viable today than ever before, because the basic processor building block is better suited to the job.

The second and perhaps more fundamental observation is that change, even dramatic change, is the norm in computer architecture. The continuing process of change has profound implications for the study of computer architecture, because we need to understand not only how things are, but how they might evolve, and why. Change is one of the key challenges in writing this book, and one of the key motivations. Parallel computer architecture has matured to the point where it needs to be studied from a basis of engineering principles and quantitative evaluation of performance and cost. These are rooted in a body of facts, measurements, and designs of real machines.

---

1. The figure is drawn from an influential 1991 paper that sought to explain the dramatic changes taking place in the computing industry[HeJo91] The metric of performance is a bit tricky when reaching across such a range of time and market segment. The study draws data from general purpose benchmarks, such as the SPEC benchmark that is widely used to assess performance on technical computing applications[HePa90]. After publication, microprocessors continued to track the prediction, while mainframes and supercomputers went through tremendous crises and emerged using multiple CMOS microprocessors in their market niche.
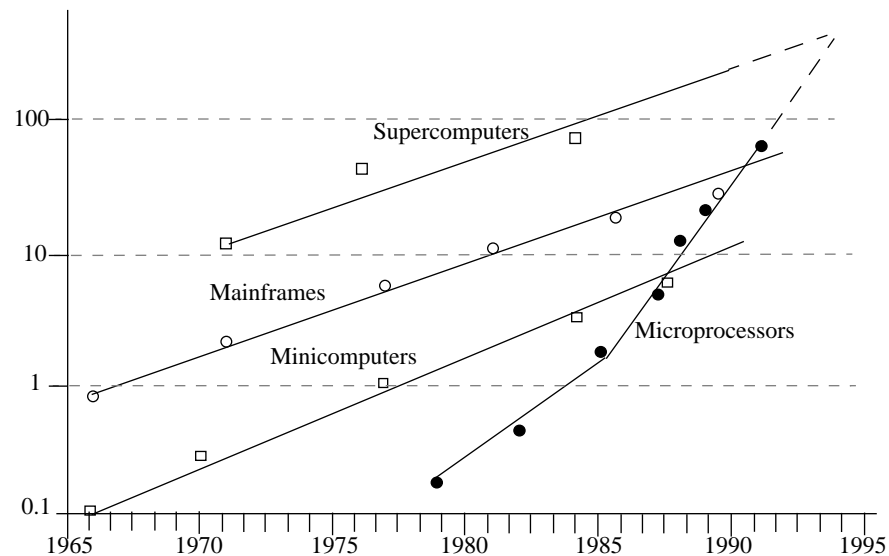
Figure 1-1 Performance trends over time of micro, mini, mainframe and supercomputer processors.[HeJo91]

Performance of microprocessors has been increasing at a rate of nearly 50% per year since the mid 80's. More traditional mainframe and supercomputer performance has been increasing at a rate of roughly 25% per year. As a result we are seeing the processor that is best suited to parallel architecture become the performance leader as well.

Unfortunately, the existing data and designs are necessarily frozen in time, and will become dated as the field progresses. We have elected to present hard data and examine real machines throughout the book in the form of a "late 1990s" technological snapshot in order to retain this grounding. We strongly believe that the methods of evaluation underlying the analysis of concrete design trade-offs transcend the chronological and technological reference point of the book.

The "late 1990s" happens to be a particularly interesting snapshot, because we are in the midst of a dramatic technological realignment as the single-chip microprocessor is poised to dominate every sector of computing, and as parallel computing takes hold in many areas of mainstream computing. Of course, the prevalence of change suggests that one should be cautious in extrapolating toward the future. In the remainder of this section, we examine more deeply the forces and trends that are giving parallel architectures an increasingly important role throughout the computing field and pushing parallel computing into the mainstream. We look first at the application demand for increased performance and then at the underlying technological and architectural trends that strive to meet these demands. We see that parallelism is inherently attractive as computers become more highly integrated, and that it is being exploited at increasingly high levels of the design. Finally, we look at the role of parallelism in the machines at the very high end of the performance spectrum.

1.2.1 Application Trends

The demand for ever greater application performance is a familiar feature of every aspect of computing. Advances in hardware capability enable new application functionality, which grows in

significance and places even greater demands on the architecture, and so on. This cycle drives the tremendous ongoing design, engineering, and manufacturing effort underlying the sustained exponential performance increase in microprocessor performance. It drives parallel architecture even harder, since parallel architecture focuses on the most demanding of these applications. With a 50% annual improvement in processor performance, a parallel machine of a hundred processors can be viewed as providing to applications the computing power that will be widely available ten years in the future, whereas a thousand processors reflects nearly a twenty year horizon.

Application demand also leads computer vendors to provide a range of models with increasing performance and capacity at progressively increased cost. The largest volume of machines and greatest number of users are at the low end, whereas the most demanding applications are served by the high end. One effect of this "platform pyramid" is that the pressure for increased performance is greatest at the high-end and is exerted by an important minority of the applications. Prior to the microprocessor era, greater performance was obtained through exotic circuit technologies and machine organizations. Today, to obtain performance significantly greater than the state-of-the-art microprocessor, the primary option is multiple processors, and the most demanding applications are written as parallel programs. Thus, parallel architectures and parallel applications are subject to the most acute demands for greater performance.

A key reference point for both the architect and the application developer is how the use of parallelism improves the performance of the application. We may define the *speedup* on p processors as

$$Speedup(p \text{ processors}) \equiv \frac{Performance(p \text{ processors})}{Performance(1 \text{ processors})}.$$ **(EQ 1.1)**

For a single, fixed problem, the performance of the machine on the problem is simply the reciprocal of the time to complete the problem, so we have the following important special case:

$$Speedup_{\text{fixed problem}}(p \text{ processors}) = \frac{Time(1 \text{ processor})}{Time(p \text{ processors})}.$$ **(EQ 1.2)**

Scientific and Engineering Computing

The direct reliance on increasing levels of performance is well established in a number of endeavors, but is perhaps most apparent in the field of computational science and engineering. Basically, computers are used to simulate physical phenomena that are impossible or very costly to observe through empirical means. Typical examples include modeling global climate change over long periods, the evolution of galaxies, the atomic structure of materials, the efficiency of combustion with an engine, the flow of air over surfaces of vehicles, the damage due to impacts, and the behavior of microscopic electronic devices. Computational modeling allows in-depth analyses to be performed cheaply on hypothetical designs through computer simulation. A direct correspondence can be drawn between levels of computational performance and the problems that can be studied through simulation. Figure 1-2 summarizes the 1993 findings of the Committee on Physical, Mathematical, and Engineering Sciences of the federal Office of Science and Technology Policy[OST93]. It indicates the computational rate and storage capacity required to tackle a number of important science and engineering problems. Also noted is the year in which this capability was forecasted to be available. Even with dramatic increases in processor performance, very large parallel architectures are needed to address these problems in the near future. Some years further down the road, new grand challenges will be in view.

Grand Challenge Problems

Global Change
Human Genome
Fluid Turbulence
Vehicle Dynamics
Ocean Circulation
Viscous Fluid Dyn's
Superconductor Mod.
QCD, Vision

Structural
Biology

Vehicle
Signature   Pharmaceutical Design

72 hour
Weather

48 Hour   3D Plasma
Weather   Modelling

Chemical Dynamics

2D   Oil Resevoir
Airfoil   Modeling

Storage Requirement:
1 TB
100 GB
10 GB
1 GB
100 MB
10 MB

Computational Performance Requirement:
100 Mflops   1 Gflops   10 Gflops   100 Gflops   1Tflops

Figure 1-2 Grand Challenge Application Requirements

A collection of important scientific and engineering problems are positioned in a space defined by computational performance and storage capacity. Given the exponential growth rate of performance and capacity, both of these axes map directly to time. In the upper right corner appears some of the Grand Challenge applications identified by the U.S. High Performance Computing and Communications program.

Parallel architectures have become the mainstay of scientific computing, including physics, chemistry, material science, biology, astronomy, earth sciences, and others. The engineering application of these tools for modeling physical phenomena is now essential to many industries, including petroleum (reservoir modeling), automotive (crash simulation, drag analysis, combustion efficiency), aeronautics (airflow analysis, engine efficiency, structural mechanics, electromagnetism), pharmaceuticals (molecular modeling), and others. In almost all of these applications, there is a large demand for visualization of the results, which is itself a demanding application amenable to parallel computing.

The visualization component has brought the traditional areas of scientific and engineering computing closer to the entertainment industry. In 1995, the first full-length computer-animated motion picture, Toy Story, was produced on a parallel computer system composed of hundreds of Sun workstations. This application was finally possible because the underlying technology and architecture crossed three key thresholds: the cost of computing dropped to where the rendering could be accomplished within the budget typically associated with a feature film, while the performance of the individual processors and the scale of parallelism rose to where it could be accomplished in a reasonable amount of time (several months on several hundred processors).

Each science and engineering applications has an analogous threshold of computing capacity and cost at which it becomes viable.

Let us take an example from the Grand Challenge program to help understand the strong interaction between applications, architecture, and technology in the context of parallel machines. A 1995 study[JNNIE] examined the effectiveness of a wide range of parallel machines on a variety of applications, including a molecular dynamics package, AMBER (Assisted Model Building through Energy Refinement). AMBER is widely used to simulate the motion of large biological models such as proteins and DNA, which consist of sequences of residues (amino acids and nucleic acids, respectively) each composed of individual atoms. The code was developed on Cray vector supercomputers, which employ custom ECL-based processors, large expensive SRAM memories, instead of caches, and machine instructions that perform arithmetic or data movement on a sequence, or *vector*, of data values. Figure 1-3 shows the speedup obtained on three versions of this code on a 128-processor microprocessor-based machine - the Intel Paragon, described later. The particular test problem involves the simulation of a protein solvated by water. This test consisted of 99 amino acids and 3,375 water molecules for approximately 11,000 atoms.

The initial parallelization of the code (vers. 8/94) resulted in good speedup for small configurations, but poor speedup on larger configurations. A modest effort to improve the balance of work done by each processor, using techniques we discuss in Chapter 2, improved the scaling of the application significantly (vers. 9/94). An additional effort to optimize communication produced a highly scalable version(12/94). This 128 processor version achieved a performance of 406 MFLOPS; the best previously achieved was 145 MFLOPS on a Cray C90 vector processor. The same application on a more efficient parallel architecture, the Cray T3D, achieved 891 MFLOPS on 128 processors. This sort of "learning curve" is quite typical in the parallelization of important applications, as is the interaction between application and architecture. The application writer typically studies the application to understand the demands it places on the available architectures and how to improve its performance on a given set of machines. The architect may study these demands as well in order to understand how to make the machine more effective on a given set of applications. Ideally, the end user of the application enjoys the benefits of both efforts.

The demand for ever increasing performance is a natural consequence of the modeling activity. For example, in electronic CAD as the number of devices on the chip increases, there is obviously more to simulate. In addition, the increasing complexity of the design requires that more test vectors be used and, because higher level functionality is incorporated into the chip, each of these tests must run for a larger number of clock cycles. Furthermore, an increasing level of confidence is required, because the cost of fabrication is so great. The cumulative effect is that the computational demand for the design verification of each new generation is increasing at an even faster rate than the performance of the microprocessors themselves.

Commercial Computing

Commercial computing has also come to rely on parallel architectures for its high-end. Although the scale of parallelism is typically not as large as in scientific computing, the use of parallelism is even more wide-spread. Multiprocessors have provided the high-end of the commercial computing market since the mid-60s. In this arena, computer system speed and capacity translate directly into the scale of business that can be supported by the system. The relationship between performance and scale of business enterprise is clearly articulated in the on-line transaction processing (OLTP) benchmarks sponsored by the Transaction Processing Performance Council
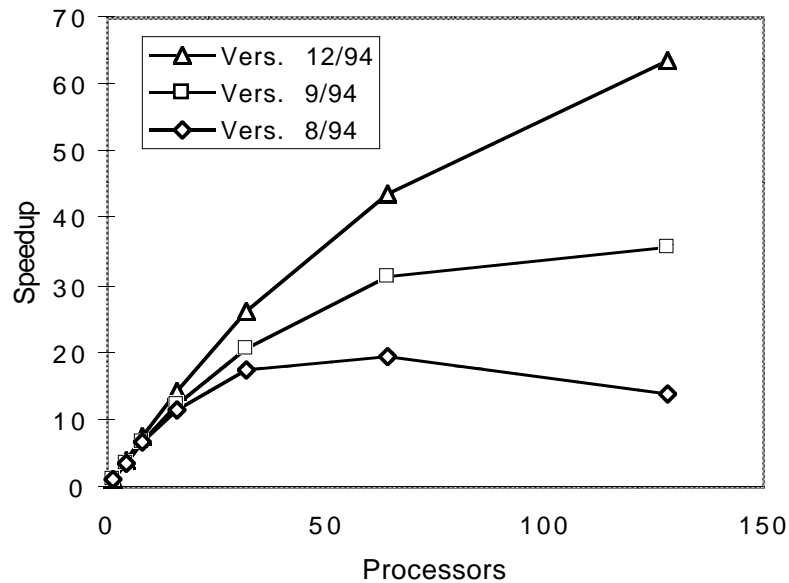
Figure 1-3 Speedup on Three Versions of a Parallel Program

The parallelization learning curve is illustrated by the speedup obtained on three successive versions of this molecular dynamics code on the Intel Paragon.

(TPC) [tpc]. These benchmarks rate the performance of a system in terms of its throughput in *transactions-per-minute* (tpm) on a typical workload. TPC-C is an order entry application with a mix of interactive and batch transactions, including realistic features like queued transactions, aborting transactions, and elaborate presentation features[GrRe93]. The benchmark includes an explicit scaling criteria to make the problem more realistic: the size of the database and the number of terminals in the system increase as the tpmC rating rises. Thus, a faster system must operate on a larger database and service a larger number of users.

Figure 1-4 shows the tpm-C ratings for the collection of systems appearing in one edition of the TPC results (March 1996), with the achieved throughput on the vertical axis and the number of processors employed in the server along the horizontal axis. This data includes a wide range of systems from a variety of hardware and software vendors; we have highlighted the data points for models from a few of the vendors. Since the problem scales with system performance, we cannot compare times to see the effectiveness of parallelism. Instead, we use the throughput of the system as the metric of performance in Equation 1.1.
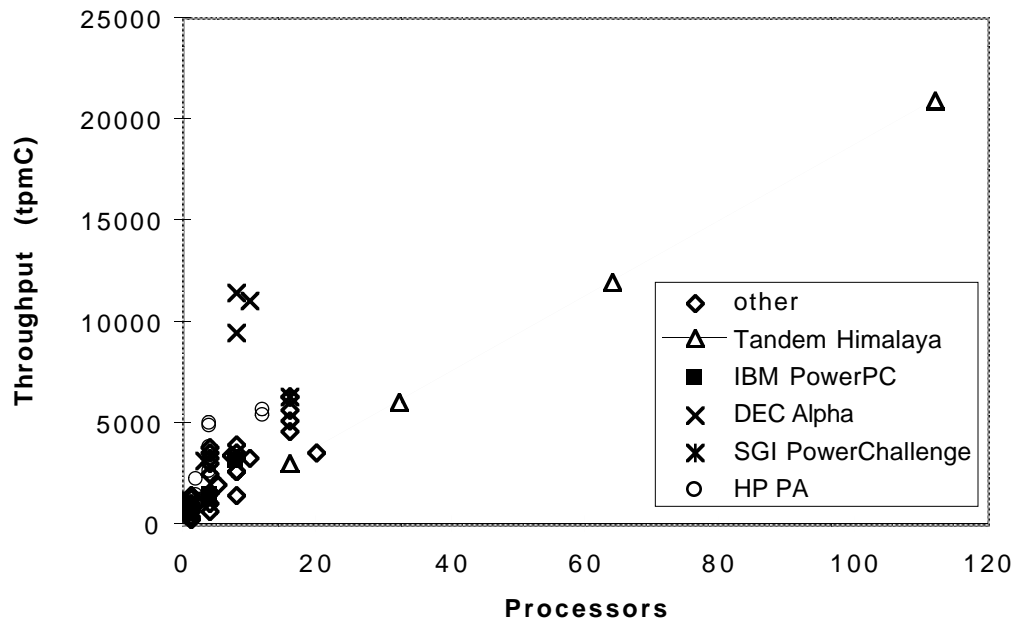
Figure 1-4 TPC-C throughput versus number of processors on TPC

he March 1996 TPC report documents the transaction processing performance for a wide range of system. The figure shows the number of processors employed for all of the high end systems, highlights five leading vendor product lines. All of the major database vendors utilize multiple processors for their high performance options, although the scale of parallelism varies considerably.

Example 1-1   The tpmC for the Tandem Himalaya and IBM Power PC systems are given in the following table. What is the speedup obtained on each?

	tpmC	
Number of Processors	IBM RS 6000 PowerPC	Himalaya K10000
1	735	
4	1438	
8	3119	
16		3043
32		6067
64		12021
112		20918

For the IBM system we may calculate speedup relative to the uniprocessor system, whereas in the Tandem case we can only calculate speedup relative to a sixteen processor system. For the IBM machine there appears to be a significant penalty in the parallel database implementation in

going from one to four processors, however, the scaling is very good (superlinear) from four to eight processors. The Tandem system achieves good scaling, although the speedup appears to be beginning to flatten towards the hundred processor regime.

Speedup$_{tpmC}$

Number of Processors	IBM RS 6000 PowerPC	Himalaya K10000
1	1	
4	1.96	
8	4.24	
16		1
32		1.99
64		3.95
112		6.87

Several important observations can be drawn from the TPC data. First, the use of parallel architectures is prevalent. Essentially all of the vendors supplying database hardware or software offer multiprocessor systems that provide performance substantially beyond their uniprocessor product. Second, it is not only large scale parallelism that is important, but modest scale multiprocessor servers with tens of processors, and even small-scale multiprocessors with two or four processors. Finally, even a set of well-documented measurements of a particular class of system at a specific point in time cannot provide a true technological snapshot. Technology evolves rapidly, systems take time to develop and deploy, and real systems have a useful lifetime. Thus, the best systems available from a collection of vendors will be at different points in their life cycle at any time. For example, the DEC Alpha and IBM PowerPC systems in the 3/96 TPC report were much newer, at the time of the report, than the Tandem Himalaya system. We cannot conclude, for example, that the Tandem system is inherently less efficient as a result of its scalable design. We can, however, conclude that even very large scale systems must track the technology to retain their advantage.

Even the desktop demonstrates a significant number of concurrent processes, with a host of active windows and daemons. Quite often a single user will have tasks running on many machines within the local area network, or farm tasks across the network. The transition to parallel programming, including new algorithms in some cases or attention to communication and synchronization requirements in existing algorithms, has largely taken place in the high performance end of computing, especially in scientific programming. The transition is in progress among the much broader base of commercial engineering software. In the commercial world, all of the major database vendors support parallel machines for their high-end products. Typically, engineering and commercial applications target more modest scale multiprocessors, which dominate the server market. However, the major database vendors also offer "shared-nothing" versions for large parallel machines and collections of workstations on a fast network, often called *clusters*. In addition, multiprocessor machines are heavily used to improve throughput on multiprogramming workloads. All of these trends provide a solid application demand for parallel architectures of a variety of scales.
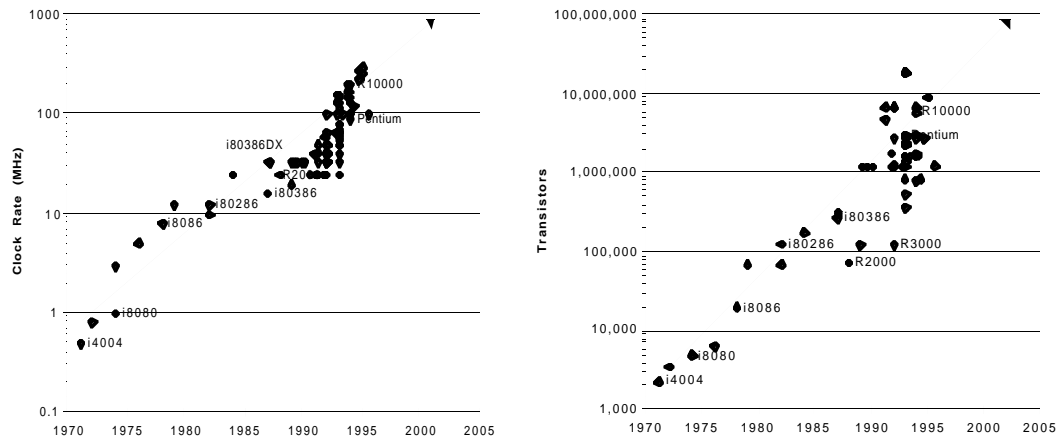
Figure 1-5 Improvement in logic density and clock frequency of microprocessors.

Improvements in lithographic technique, process technology, circuit design, and datapath design have yielded a sustained improvement in logic density and clock rate.

1.2.2   Technology Trends

The importance of parallelism in meeting the application demand for ever greater performance can be brought into sharper focus by looking more closely at the advancements in the underlying technology and architecture. These trends suggest that it may be increasingly difficult to "wait for the single processor to get fast enough," while parallel architectures will become more and more attractive. Moreover, the examination shows that the critical issues in parallel computer architecture are fundamentally similar to those that we wrestle with in "sequential" computers, such as how the resource budget should be divided up among functional units that do the work, caches to exploit locality, and wires to provide bandwidth.

The primary technological advance is a steady reduction in the basic VLSI feature size. This makes transistors, gates, and circuits faster and smaller, so more fit in the same area. In addition, the useful die size is growing, so there is more area to use. Intuitively, clock rate improves in proportion to the improvement in feature size, while the number of transistors grows as the square, or even faster due to increasing overall die area. Thus, in the long run the use of many transistors at once, *i.e.*, parallelism, can be expected to contribute more than clock rate to the observed performance improvement of the single-chip building block.

This intuition is borne out by examination of commercial microprocessors. Figure 1-5 shows the increase in clock frequency and transistor count for several important microprocessor families. Clock rates for the leading microprocessors increase by about 30% per year, while the number of transistors increases by about 40% per year. Thus, if we look at the raw computing power of a chip (total transistors switching per second), transistor capacity has contributed an order of magnitude more than clock rate over the past two decades.[1] The performance of microprocessors on standard benchmarks has been increasing at a much greater rate. The most widely used benchmark for measuring workstation performance is the SPEC suite, which includes several realistic

integer programs and floating point programs[SPEC]. Integer performance on SPEC has been increasing at about 55% per year, and floating-point performance at 75% per year. The LINPACK benchmark[Don94] is the most widely used metric of performance on numerical applications. LINPACK floating-point performance has been increasing at more than 80% per year. Thus, processors are getting faster in large part by making more effective use of an ever larger volume of computing resources.

The simplest analysis of these technology trends suggests that the basic single-chip building block will provide increasingly large capacity, in the vicinity of 100 million transistors by the year 2000. This raises the possibility of placing more of the computer system on the chip, including memory and I/O support, or of placing multiple processors on the chip[Gwe94b]. The former yields a small and conveniently packaged building block for parallel architectures. The latter brings parallel architecture into the single chip regime[Gwe94a]. Both possibilities are in evidence commercially, with the system-on-a-chip becoming first established in embedded systems, portables, and low-end personal computer products.[x86,mips] Multiple processors on a chip is becoming established in digital signal processing[Fei94].

The divergence between capacity and speed is much more pronounced in memory technology. From 1980 to 1995, the capacity of a DRAM chip increased a thousand-fold, quadrupling every three years, while the memory cycle time improved by only a factor of two. In the time-frame of the 100 million transistor microprocessor, we anticipate gigabit DRAM chips, but the gap between processor cycle time and memory cycle time will have grown substantially wider. Thus, the memory bandwidth demanded by the processor (bytes per memory cycle) is growing rapidly and in order to keep pace, we must transfer more data in parallel. From PCs to workstations to servers, designs based on conventional DRAMs are using wider and wider paths into the memory and greater interleaving of memory banks. Parallelism. A number of advanced DRAM designs are appearing on the market which transfer a large number of bits per memory cycle within the chip, but then pipeline the transfer of those bits across a narrower interface at high frequency. In addition, these designs retain recently data in fast on-chip buffers, much as processor caches do, in order to reduce the time for future accesses. Thus, exploiting parallelism and locality is central to the advancements in memory devices themselves.

The latency of a memory operation is determined by the access time, which is smaller than the cycle time, but still the number of processor cycles per memory access time is large and increasing. To reduce the average latency experienced by the processor and to increase the bandwidth that can be delivered to the processor, we must make more and more effective use of the levels of the memory hierarchy that lie between the processor and the DRAM memory. As the size of the memory increases, the access time increases due to address decoding, internal delays in driving long bit lines, selection logic, and the need to use a small amount of charge per bit. Essentially all modern microprocessors provide one or two levels of caches on chip, plus most system designs provide an additional level of external cache. A fundamental question as we move into multiprocessor designs is how to organize the collection of caches that lie between the many processors and the many memory modules. For example, one of the immediate benefits of parallel architec-

---

1. There are many reasons why the transistor count does not increase as the square of the clock rate. One is that much of the area of a processor is consumed by wires, serving to distribute control, data, or clock, i.e., on-chip communication. We will see that the communication issue reappears at every level of parallel computer architecture.

tures is that the total size of each level of the memory hierarchy can increase with the number of processors without increasing the access time.

Extending these observations to disks, we see a similar divergence. Parallel disks storage systems, such as RAID, are becoming the norm. Large, multi-level caches for files or disk blocks are predominant.

1.2.3 Architectural Trends

Advances in technology determine what is possible; architecture translates the potential of the technology into performance and capability. There are fundamentally two ways in which a larger volume of resources, more transistors, improves performance: *parallelism* and *locality*. Moreover, these two fundamentally compete for the same resources. Whenever multiple operations are performed in parallel the number of cycles required to execute the program is reduced. However, resources are required to support each of the simultaneous activities. Whenever data references are performed close to the processor, the latency of accessing deeper levels of the storage hierarchy is avoided and the number of cycles to execute the program is reduced. However, resources are also required to provide this local storage. In general, the best performance is obtained by an intermediate strategy which devotes resources to exploit a degree of parallelism and a degree of locality. Indeed, we will see throughout the book that parallelism and locality interact in interesting ways in systems of all scales, from within a chip to across a large parallel machine. In current microprocessors, the die area is divided roughly equally between cache storage, processing, and off-chip interconnect. Larger scale systems may exhibit a somewhat different split, due to differences in the cost and performance trade-offs, but the basic issues are the same.

Examining the trends in microprocessor architecture will help build intuition towards the issues we will be dealing with in parallel machines. It will also illustrate how fundamental parallelism is to conventional computer architecture and how current architectural trends are leading toward multiprocessor designs. (The discussion of processor design techniques in this book is cursory, since many of the readers are expected to be familiar with those techniques from traditional architecture texts[HeP90] or the many discussions in the trade literature. It does provide provide an unique perspective on those techniques, however, and will serve to refresh your memory.)

The history of computer architecture has traditionally been divided into four generations identified by the basic logic technology: tubes, transistors, integrated circuits, and VLSI. The entire period covered by the figures in this chapter is lumped into the fourth or VLSI generation. Clearly, there has been tremendous architectural advance over this period, but what delineates one era from the next within this generation? The strongest delineation is the kind of parallelism that is exploited. The period up to about 1985 is dominated by advancements in *bit-level parallelism*, with 4-bit microprocessors replaced by 8-bit, 16-bit, and so on. Doubling the width of the datapath reduces the number of cycles required to perform a full 32-bit operation. This trend slows once a 32-bit word size is reached in the mid-80s, with only partial adoption of 64-bit operation obtained a decade later. Further increases in word-width will be driven by demands for improved floating-point representation and a larger address space, rather than performance. With address space requirements growing by less than one bit per year, the demand for 128-bit operation appears to be well in the future. The early microprocessor period was able to reap the benefits of the easiest form of parallelism: bit-level parallelism in every operation. The dramatic inflection point in the microprocessor growth curve in Figure 1-1 marks the arrival of full 32-bit word operation combined with the prevalent use of caches.
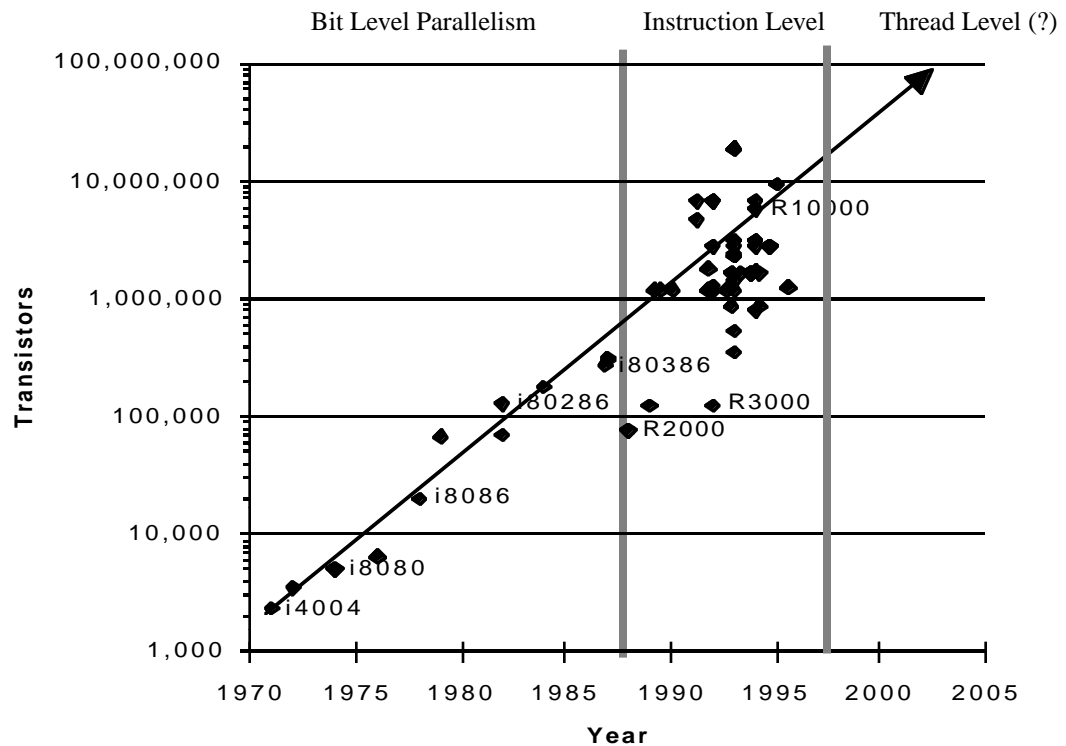
Figure 1-6 Number of transistors per processor chip over the last 25 years.

The growth essentially follows Moore's law which says that the number of transistors doubles every 2 years. Forecasting from past trends we can reasonably expect to be designing for a 50-100 million transistor budget at the end of the decade. Also indicated are the epochs of design within the fourth, or VLSI generation of computer architecture reflecting the increasing level of parallelism.

The period from the mid-80s to mid-90s is dominated by advancements in *instruction-level parallelism*. Full word operation meant that the basic steps in instruction processing (instruction decode, integer arithmetic, and address calculation) could be performed in a single cycle; with caches the instruction fetch and data access could also be performed in a single cycle, most of the time. The RISC approach demonstrated that, with care in the instruction set design, it was straightforward to pipeline the stages of instruction processing so that an instruction is executed almost every cycle, on average. Thus the parallelism inherent in the steps of instruction processing could be exploited across a small number of instructions. While pipelined instruction processing was not new, it had never before been so well suited to the underlying technology. In addition, advances in compiler technology made instruction pipelines more effective.

The mid-80s microprocessor-based computers consisted of a small constellation of chips: an integer processing unit, a floating-point unit, a cache controller, and SRAMs for the cache data and tag storage. As chip capacity increased these components were coalesced into a single chip, which reduced the cost of communicating among them. Thus, a single chip contained separate hardware for integer arithmetic, memory operations, branch operations, and floating-point operations. In addition to pipelining individual instructions, it became very attractive to fetch multiple instructions at a time and issue them in parallel to distinct function units whenever possible. This

form of instruction level parallelism came to be called *superscalar* execution. It provided a natural way to exploit the ever increasing number of available chip resources. More function units were added, more instructions were fetched at time, and more instructions could be issued in each clock cycle to the function units.

However, increasing the amount of instruction level parallelism that the processor can exploit is only worthwhile if the processor can be supplied with instructions and data fast enough to keep it busy. In order to satisfy the increasing instruction and data bandwidth requirement, larger and larger caches were placed on-chip with the processor, further consuming the ever increasing number of transistors. With the processor and cache on the same chip, the path between the two could be made very wide to statisfy the bandwidth requirement of multiple instruction and data accesses per cycle. However, as more instructions are issued each cycle, the performance impact of each control transfer and each cache miss becomes more significant. A control transfer may have to wait for the depth, or *latency*, of the processor pipeline, until a particular instruction reaches the end of the pipeline and determines which instruction to execute next. Similarly, instructions which use a value loaded from memory may cause the processor to wait for the latency of a cache miss.

Processor designs in the 90s deploy a variety of complex instruction processing mechanisms in an effort to reduce the performance degradation due to latency in "wide-issue" superscalar processors. Sophisticated branch prediction techniques are used to avoid pipeline latency by guessing the direction of control flow before branches are actually resolved. Larger, more sophisticated caches are used to *avoid* the latency of cache misses. Instructions are scheduled dynamically and allowed to complete out of order so if one instruction encounters a miss, other instructions can proceed ahead of it, as long as they do not depend on the result of the instruction. A larger window of instructions that are waiting to issue is maintained within the processor and whenever an instruction produces a new result, several waiting instructions may be issued to the function units. These complex mechanisms allow the processor to *tolerate* the latency of a cache-miss or pipeline dependence when it does occur. However, each of these mechanisms place a heavy demand on chip resources and a very heavy design cost.

Given the expected increases in chip density, the natural question to ask is how far will instruction level parallelism go within a single thread of control? At what point will the emphasis shift to supporting the higher levels of parallelism available as multiple processes or multiple threads of contol within a process, i.e, *thread level parallelism?* Several research studies have sought to answer the first part of the question, either through simulation of aggressive machine designs[Cha*91,Hor*90,Lee*91,MePa91] or through analysis of the inherent properties of programs[But*91,JoWa89,Joh90,Smi*89,Wal91]. The most complete treatment appears in Johnson's book devoted to the topic[Joh90]. Simulation of aggressive machine designs generally shows that 2-way superscalar, i.e., issuing two instructions per cycle, is very profitable and 4-way offers substantial additional benefit, but wider issue widths, e.g., 8-way superscalar, provide little additional gain. The design complexity increases dramatically, because control transfers occur roughly once in five instructions, on average.

To estimate the maximum potential speedup that can be obtained by issuing multiple instructions per cycle, the execution trace of a program is simulated on an ideal machine with unlimited instruction fetch bandwidth, as many functions units as the program can use, and perfect branch prediction. (The latter is easy, since the trace correctly follows each branch.) These generous machine assumptions ensure that no instruction is held up because a function unit is busy or because the instruction is beyond the look-ahead capability of the processor. Furthermore, to

ensure that no instruction is delayed because it updates a location that is used by logically previous instructions, storage resource dependences are removed by a technique called *renaming*. Each update to a register or memory location is treated as introducing a new "name," and subsequent uses of the value in the execution trace refer to the new name. In this way, the execution order of the program is constrained only by essential data dependences; each instruction is executed as soon as its operands are available. Figure 1-7 summarizes the result of this "ideal machine" analysis based on data presented by Johnson[Joh91]. The histogram on the left shows the fraction of cycles in which no instruction could issue, only one instruction, and so on. Johnson's ideal machine retains realistic function unit latencies, including cache misses, which accounts for the zero-issue cycles. (Other studies ignore cache effects or ignore pipeline latencies, and thereby obtain more optimistic estimates.) We see that even with infinite machine resources, perfect branch prediction, and ideal renaming, 90% of the time no more than four instructions issue in a cycle. Based on this distribution, we can estimate the speedup obtained at various issue widths, as shown in the right portion of the figure. Recent work[LaWi92,Soh94] provides empirical evidence that to obtain significantly larger amounts of parallelism, multiple threads of control must be pursued simultaneously. Barring some unforeseen breakthrough in instruction level parallelism, the leap to the next level of useful parallelism, multiple concurrent threads, is increasingly compelling as chips increase in capacity.
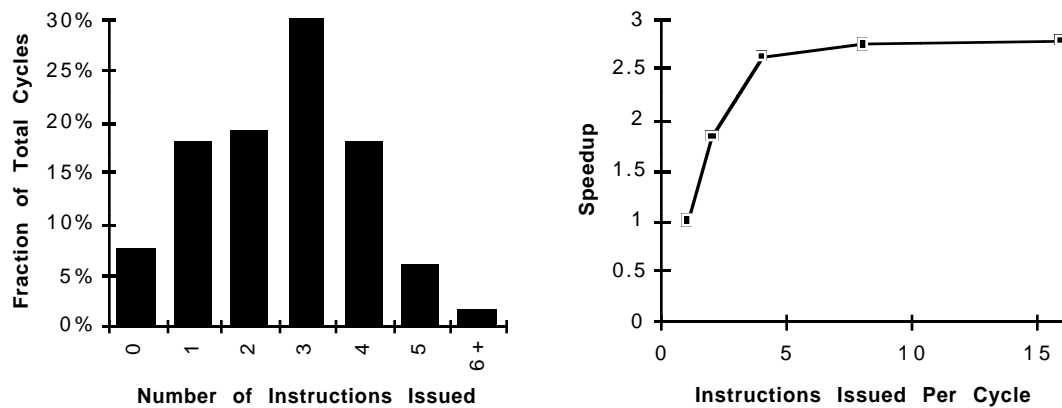
Figure 1-7 Distribution of potential instruction-level parallelism and estimated speedup under ideal superscalar execution

The figure shows the distribution of available instruction-level parallelism and maximum potential speedup under idealized superscalar execution, including unbounded processing resources and perfect branch prediction. Data is an average of that presented for several benchmarks by Johnson[Joh91].

The trend toward thread or process-level parallelism has been strong at the computer system level for some time. Computers containing multiple state-of-the-art microprocessors sharing a common memory became prevalent in the mid 80's, when the 32-bit microprocessor was first introduced[Bel85]. As indicated by Figure 1-8, which shows the number of processors available in commercial multiprocessors over time, this bus-based shared-memory multiprocessor approach has maintained a substantial multiplier to the increasing performance of the individual processors. Almost every commercial microprocessor introduced since the mid-80s provides hardware support for multiprocessor configurations, as we discuss in Chapter 5. Multiprocessors dominate the server and enterprise (or mainframe) markets and have migrated down to the desktop.

The early multi-microprocessor systems were introduced by small companies competing for a share of the minicomputer market, including Synapse[Nes85], Encore[Sha85], Flex[Mat85], Sequent[Rod85] and Myrias[Sav85]. They combined 10 to 20 microprocessors to deliver competitive throughput on timesharing loads. With the introduction of the 32-bit Intel i80386 as the base processor, these system obtained substantial commercial success, especially in transaction processing. However, the rapid performance advance of RISC microprocessors, exploiting instruction level parallelism, sapped the CISC multiprocessor momentum in the late 80s (and all but eliminated the minicomputer). Shortly thereafter, several large companies began producing RISC multiprocessor systems, especially as servers and mainframe replacements. Again, we see the critical role of bandwidth. In most of these multiprocessor designs, all the processors plug into a common bus. Since a bus has a fixed aggregate bandwidth, as the processors become faster, a smaller number can be supported by the bus. The early 1990s brought a dramatic advance in the shared memory bus technology, including faster electrical signalling, wider data paths, pipelined protocols, and multiple paths. Each of these provided greater bandwidth, growing with time and design experience, as indicated in Figure 1-9. This allowed the multiprocessor designs to ramp back up to the ten to twenty range and beyond, while tracking the microprocessor advances[AR94,Cek*93,Fen*95,Fra93,Gal*94,GoMa95].

The picture in the mid-90s is very interesting. Not only has the bus-based shared-memory multiprocessor approach become ubiquitous in the industry, it is present at a wide range of scale. Desktop systems and small servers commonly support two to four processors, larger servers support tens, and large commercial systems are moving toward a hundred. Indications are that this trend will continue. As indication of the shift in emphasis, in 1994 Intel defined a standard approach to the design of multiprocessor PC systems around its Pentium microprocessor[Sla94]. The follow-on PentiumPro microprocessor allows four-processor configurations to be constructed by wiring the chips together without even any glue logic; bus drivers, arbitration, and so on are in the microprocessor. This development is expected to make small-scale multiprocessors a true commodity. Additionally, a shift in the industry business model has been noted, where multiprocessors are being pushed by software vendors, especially database companies, rather than just by the hardware vendors. Combining these trends with the technology trends, it appears that the question is when, not if, multiple processors per chip will become prevalent.

1.2.4 Supercomputers

We have looked at the forces driving the development of parallel architecture in the general market. A second, confluent set of forces come from the quest to achieve absolute maximum performance, or *supercomputing*. Although commercial and information processing applications are increasingly becoming important drivers of the high end, historically, scientific computing has been a kind of proving ground for innovative architecture. In the mid 60's this included pipelined instruction processing and dynamic instruction scheduling, which are commonplace in microprocessors today. Starting in the mid 70's, supercomputing was dominated by *vector processors,* which perform operations on sequences of data elements, i.e, a vector, rather than individual scalar data. Vector operations permit more parallelism to be obtained within a single thread of control. Also, these vector supercomputers were implemented in very fast, expensive, high power circuit technologies.

Dense linear algebra is an important component of Scientific computing and the specific emphasis of the LINPACK benchmark. Although this benchmark evaluates a narrow aspect of system performance, it is one of the few measurements available over a very wide class of machines over
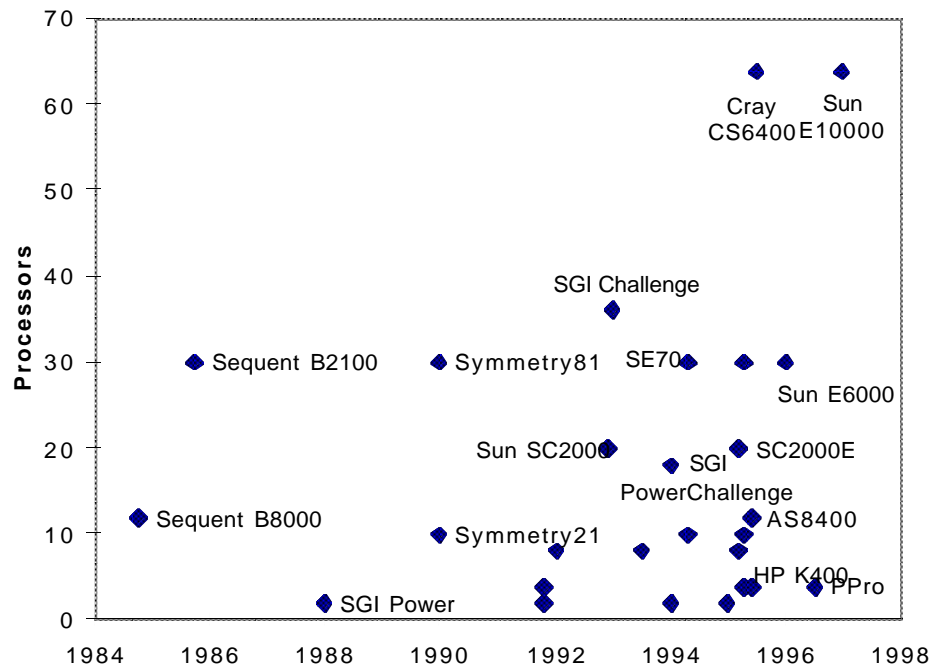
Figure 1-8 Number of processors in fully configured commercial bus-based shared-memory multiprocessors.

After an initial era of 10 to 20-way SMPs based on slow CISC microprocessors, companies such as Sun, HP, DEC, SGI, IBM and CRI began producing sizable RISC-based SMPs, as did commercial vendors not shown here, including NCR/ATT, Tandem, and Pyramid.

a long period of time. Figure 1-10 shows the Linpack performance trend for one processor of the leading Cray Research vector supercomputers[Aug*89,Rus78] compared with that of the fastest contemporary microprocessor-based workstations and servers. For each system two data points are provided. The lower one is the performance obtained on a 100x100 matrix and the higher one on a 1000x1000 matrix. Within the vector processing approach, the single processor performance improvement is dominated by modest improvements in cycle time and more substantial increases in the vector memory bandwidth. In the microprocessor systems, we see the combined effect of increasing clock rate, on-chip pipelined floating-point units, increasing on-chip cache size, increasing off-chip second-level cache size, and increasing use of instruction level parallelism.The gap in uniprocessor performance is rapidly closing.

Multiprocessor architectures are adopted by both the vector processor and microprocessor designs, but the scale is quite different. The Cray Xmp provided first two and later four processors, the Ymp eight, the C90 sixteen, and the T94 thirty-two. The microprocessor based supercomputers provided initially about a hundred processors, increasing to roughly a thousand from 1990 onward. These *massively parallel processors* (MPPs) have tracked the microprocessor advance, with typically a lag of one to two years behind the leading microprocessor-based workstation or personal computer. As shown in Figure 1-11, the large number of slightly slower microprocessors has proved dominant for this benchmark. (Note the change of scale from MFLOPS Figure 1-10to GFLOPS.) The performance advantage of the MPP systems over tradi-
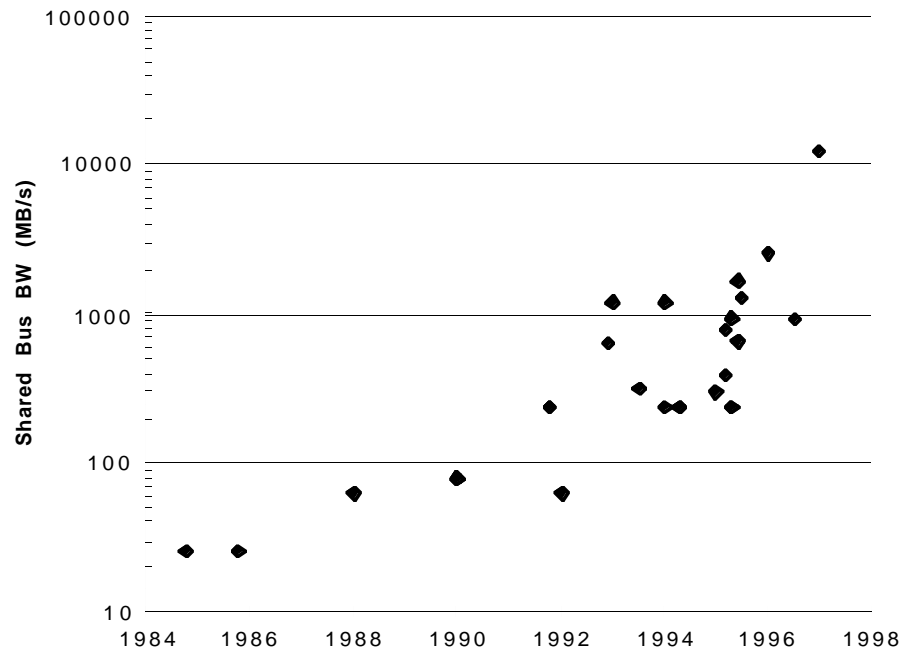
Figure 1-9 Bandwidth of the shared memory bus in commercial multiprocessors.

.After slow growth for several years, a new era of memory bus design began in 1991. This supported the use of substantial numbers of very fast microprocessors.

tional vector supercomputers is less substantial on more complete applications[NAS] owing to the relative immaturity of the programming languages, compilers, and algorithms, however, the trend toward the MPPs is still very pronounced. The importance of this trend was apparent enough that in 1993 Cray Research announced its T3D, based on the DEC Alpha microprocessor. Recently the Linpack benchmark has been used to rank the fastest computers systems in the world. Figure 1-12 shows the number of multiprocessor vector processors (PVP), MPPs, and bus-based shared memory machines (SMP) appearing in the list of the top 500 systems. The latter two are both microprocessor based and the trend is clear.

1.2.5 Summary

In examining current trends from a variety of perspectives – economics, technology, architecture, and application demand – we see that parallel architecture is increasingly attractive and increasingly central. The quest for performance is so keen that parallelism is being exploited at many different levels at various points in the computer design space. Instruction level parallelism is exploited in all modern high-performance processors. Essentially all machines beyond the desktop are multiprocessors, including servers, mainframes, and supercomputers. The very high-end of the performance curve is dominated by massively parallel processors. The use of large scale parallelism in applications is broadening and small-scale multiprocessors are emerging on the
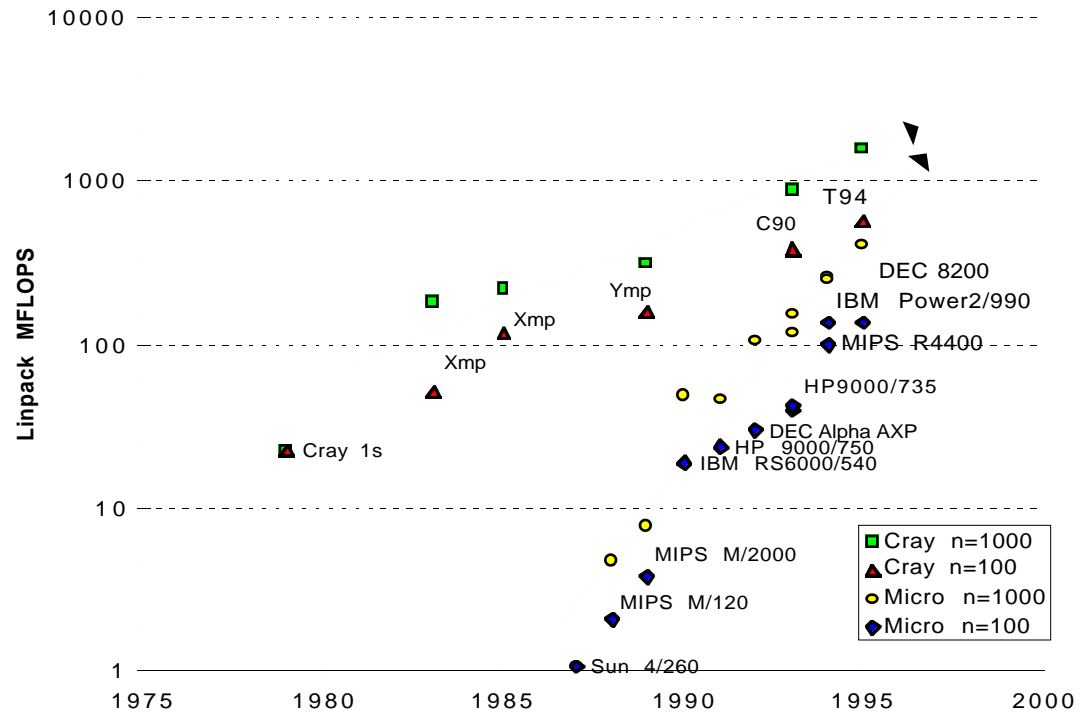
Figure 1-10 Uniprocessor performance of supercomputers and microprocessor-based systems on the LINPACK benchmark.

Performance in MFLOPS for a single processor on solving dense linear equations is shown for the leading Cray vector supercomputer and the fastest workstations on a 100x100 and 1000x1000 matrix.

desktop and servers are scaling to larger configurations. The focus of this book is the multiprocessor level of parallelism. We study the design principles embodied in parallel machines from the modest scale to the very large, so that we may understand the spectrum of viable parallel architectures that can be built from well proven components.

Our discussion of the trends toward parallel computers has been primarily from the processor perspective, but one may arrive at the same conclusion from the memory system perspective. Consider briefly the design of a memory system to support a very large amount of data, i.e., the data set of *large* problems. One of the few physical laws of computer architecture is that fast memories are small, large memories are slow. This occurs as a result of many factors, including the increased address decode time, the delays on the increasingly long bit lines, the small drive of increasingly dense storage cells, and the selector delays. This is why memory systems are constructed as a hierarchy of increasingly larger and slower memories. On average, a large hierarchical memory is fast, as long as the references exhibit good locality. The other trick we can play to "cheat the laws of physics" and obtain fast access on a very large data set is to replicate the processor and have the different processors access independent smaller memories. Of course, physics is not easily fooled. We pay the cost when a processor accesses non-local data, which we call communication, and when we need to orchestrate the actions of the many processors, *i.e.*, in synchronization operations.
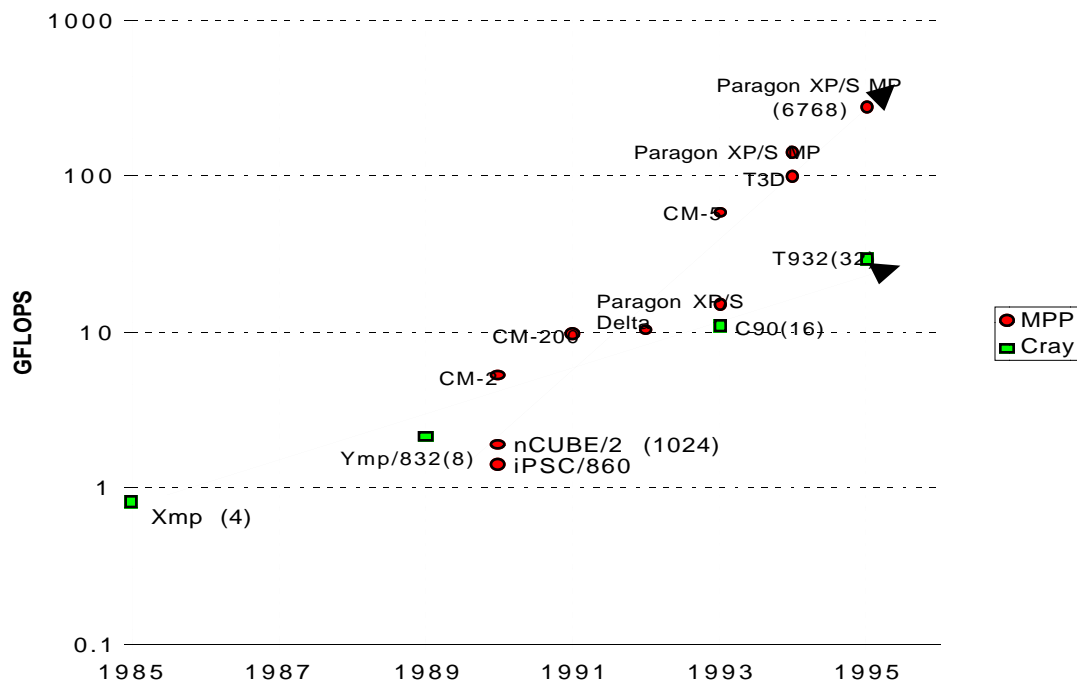
Figure 1-11 Performance of supercomputers and MPPs on the LINPACK Peak-performance benchmark.

Peak performance in GFLOPS for solving dense linear equations is shown for the leading Cray multiprocessor vector supercomputer and the fastest MPP systems. Note the change in scale from Figure 1-10.

1.3 Convergence of Parallel Architectures

Historically, parallel machines have developed within several distinct architectural "camps" and most texts on the subject are organized around a taxonomy of these designs. However, in looking at the evolution of parallel architecture, it is clear that the designs are strongly influenced by the same technological forces and similar application requirements. It is not surprising that there is a great deal of convergence in the field. In this section, our goal is to construct a framework for understanding the entire spectrum of parallel computer architectures and to build intuition as to the nature of the convergence in the field. Along the way, we will provide a quick overview of the evolution of parallel machines, starting from the traditional camps and moving toward the point of convergence.

1.3.1 Communication Architecture

Given that a parallel computer is "a collection of processing elements that communicate and cooperate to solve large problems fast," we may reasonably view parallel architecture as the extension of conventional computer architecture to address issues of communication and cooperation among processing elements. In essence, parallel architecture extends the usual concepts of a computer architecture with a *communication architecture*. Computer architecture has two dis-
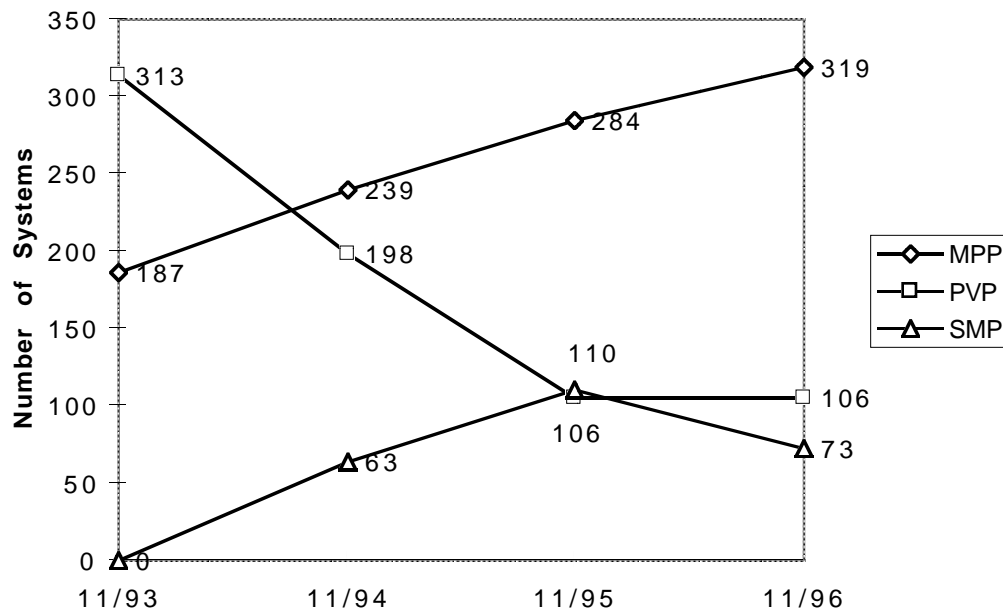
Figure 1-12 Type of systems used in 500 fastest computer systems in the world.

Parallel vector processors (PVPs) have given way to microprocessor-based Massively Parallel Processors (MPPs) and bus-based symmetric shared-memory multiprocessors (SMPs) at the high-end of computing.

tinct facets. One is the definition of critical abstractions, especially the hardware/software boundary and the user/system boundary. The architecture specifies the set of operations at the boundary and the data types that these operate on. The other facet is the organizational structure that realizes these abstractions to deliver high performance in a cost-effective manner. A communication architecture has these two facets, as well. It defines the basic communication and synchronization operations, and it addresses the organizational structures that realize these operations.

The framework for understanding communication in a parallel machine is illustrated in Figure 1-13. The top layer is the programming model, which is the conceptualization of the machine that the programmer uses in coding applications. Each programming model specifies how parts of the program running in parallel communicate information to one another and what synchronization operations are available to coordinate their activities. Applications are written in a programming model. In the simplest case, a multiprogramming workload in which a large number of independent sequential programs are run on a parallel machine, there is no communication or cooperation at the programming level. The more interesting cases include parallel programming models, such as *shared address space*, *message passing*, and *data parallel* programming. We can describe these models intuitively as follows:

- *Shared address* programming is like using a bulletin board, where one can communicate with one or many colleagues by posting information at known, shared locations. Individual activities can be orchestrated by taking note of who is doing what task.

- *Message passing* is akin to telephone calls or letters, which convey information from a specific sender to a specific receiver. There is a well-defined event when the information is sent or received, and these events are the basis for orchestrating individual activities. However, there is no shared locations accessible to all.

- *Data parallel* processing is a more regimented form of cooperation, where several agents perform an action on separate elements of a data set simultaneously and then exchange information globally before continuing *en masse*. The global reorganization of data may be accomplished through accesses to shared addresses or messages, since the programming model only defines the overall effect of the parallel steps.

A more precise definition of these programming models will be developed later in the text; at this stage it is most important to understand the layers of abstraction.

A programming model is realized in terms of the user-level communication primitives of the system, which we call the *communication abstraction*. Typically, the programming model is embodied in a parallel language or programming environment, so there is a mapping from the generic language constructs to the specific primitives of the system. These user-level primitives may be provided directly by the hardware, by the operating system or by machine specific user software which maps the communication abstractions to the actual hardware primitives. The distance between the lines in the figure is intended to indicate that the mapping from operations in the programming model to the hardware primitives may be very simple or it may be very involved. For example, access to a shared location is realized directly by load and store instructions on a machine in which all processors use the same physical memory, however, passing a message on such a machine may involve a library or system call to write the message into a buffer area or to read it out. We will examine the mapping between layers more below.
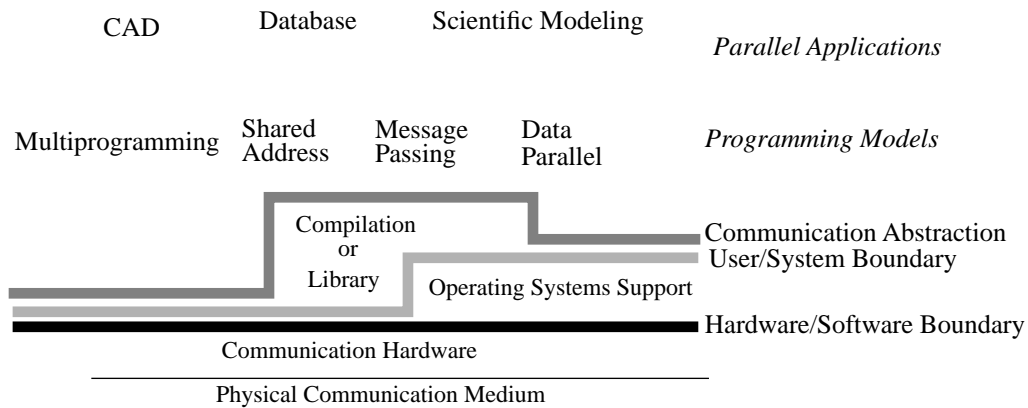
Figure 1-13 Layers of Abstraction in Parallel Computer Architecture

Critical layers of abstractions lie between the application program and the actual hardware. The application is written for a programming model, which dictates how pieces of the program share information and coordinate their activies. The specific operations providing communication and synchronization form the communication abstraction, which is he boundary between the user program and the system implementation. This abstraction is realized through compiler or library support using the primitives available from the hardware of from the operating system, which uses priviledged hardware primtives. The communication hardware is organized to provide these operations efficiently on the physical wires connecting together the machine.

The communication architecture defines the set of communication operations available to the user software, the format of these operations, and the data types they operate on, much as an

instruction set architecture does for a processor. Note that even in conventional instruction sets, some operations may be realized by a combination of hardware and software, such as a load instruction which relies on operating system intervention in the case of a page fault. The communication architecture also extends the computer organization with the hardware structures that support communication.

As with conventional computer architecture, there has been a great deal of debate over the years as to what should be incorporated into each layer of abstraction in parallel architecture and how large the "gap" should be between the layers. This debate has been fueled by various assumptions about the underlying technology and more qualitative assessments of "ease of programming." We have drawn the hardware/software boundary as flat, which might indicate that the available hardware primitives in different designs have more or less uniform complexity. Indeed, this is becoming more the case as the field matures. In most early designs the physical hardware organization was strongly oriented toward a particular programming model, *i.e.,* the communication abstraction supported by the hardware was essentially identical to the programming model. This "high level" parallel architecture approach resulted in tremendous diversity in the hardware organizations. However, as the programming models have become better understood and implementation techniques have matured, compilers and run-time libraries have grown to provide an important bridge between the programming model and the underlying hardware. Simultaneously, the technological trends discussed above have exerted a strong influence, regardless of programming model. The result has been a convergence in the organizational structure with relatively simple, general purpose communication primitives.

In the remainder of this section surveys the most widely used programming models and the corresponding styles of machine design in past and current parallel machines. Historically, parallel machines were strongly tailored to a particular programming model, so it was common to lump the programming model, the communication abstraction, and the machine organization together as "the architecture", e.g., a shared memory architecture, a message passing architecture, and so on. This approach is less appropriate today, since there is a large commonality across parallel machines and many machines support several programming models. It is important to see how this convergence has come about, so we will begin from the traditional perspective and look at machine designs associated with particular programming models, and explain their intended role and the technological opportunities that influenced their design. The goal of the survey is not to develop a taxonomy of parallel machines *per se*, but to identify a set of core concepts that form the basis for assessing design trade-offs across the entire spectrum of potential designs today and in the future. It also demonstrates the influence that the dominant technological direction established by microprocessor and DRAM technologies has had on parallel machine design, which makes a common treatment of the fundamental design issues natural or even imperative. Specifically, shared-address, message passing, data parallel, dataflow and systolic approaches are presented. In each case, we explain the abstraction embodied in the programming model and look at the motivations for the particular style of design, as well as the intended scale and application. The technological motivations for the approach are examine and how these changed over time. These changes are reflected in the machine organization, which determines what is fast and what is slow. The performance characteristics ripple up to influence aspects of the programming model. The outcome of this brief survey is a clear organizational convergence, which is captured in a generic parallel machine at then end of the section.

1.3.2 Shared Memory

One of the most important classes of parallel machines is *shared memory multiprocessors.* The key property of this class is that communication occurs implicitly as a result of conventional memory access instructions, *i.e.*, loads and stores. This class has a long history, dating at least to precursors of mainframes in the early 60s[1], and today it has a role in almost every segment of the computer industry. Shared memory multiprocessors serve to provide better throughput on multi-programming workloads, as well as to support parallel programs. Thus, they are naturally found across a wide range of scale, from a few processors to perhaps hundreds. Below we examine the communication architecture of shared-memory machines and the key organizational issues for small-scale designs and large configurations.

The primary programming model for these machine is essentially that of timesharing on a single processor, except that real parallelism replaces interleaving in time. Formally, a process is a virtual address space and one or more threads of control. Processes can be configured so that portions of their address space are shared, *i.e.*, are mapped to common physical location, as suggested by Figure 1-19. Multiple threads within a process, by definition, share portions of the address space. Cooperation and coordination among threads is accomplished by reading and writing shared variables and pointers referring to shared addresses. *Writes to a logically shared address by one thread are visible to reads of the other threads.* The communication architecture employs the conventional memory operations to provide communication through shared addresses, as well as special atomic operations for synchronization. Completely independent processes typically share the kernel portion of the address space, although this is only accessed by operating system code. Nonetheless, the shared address space model is utilized within the operating system to coordinate the execution of the processes.

While shared memory can be used for communication among arbitrary collections of processes, most parallel programs are quite structured in their use of the virtual address space. They typically have a common code image, private segments for the stack and other private data, and shared segments that are in the same region of the virtual address space of each process or thread of the program. This simple structure implies that the private variables in the program are present in each process and that shared variables have the same address and meaning in each process or thread. Often straightforward parallelization strategies are employed; for example, each process may perform a subset of the iterations of a common parallel loop or, more generally, processes may operate as a pool of workers obtaining work from a shared queue. We will discuss the structure of parallel program more deeply in Chapter 2. Here we look at the basic evolution and development of this important architectural approach.

The communication hardware for shared-memory multiprocessors is a natural extension of the memory system found in most computers. Essentially all computer systems allow a processor and a set of I/O controllers to access a collection of memory modules through some kind of hardware interconnect, as illustrated in Figure 1-15. The memory capacity is increased simply by adding memory modules. Additional capacity may or may not increase the available memory bandwidth, depending on the specific system organization. I/O capacity is increased by adding

---

1. Some say that BINAC was the first multiprocessors, but it was intended to improve reliability. The two processors check each other at every instruction. They never agreed, so people eventually turned one of them off.

Virtual address spaces for a
collection of processes communicating
via *Shared Addresses*

Machine Physical Address Space
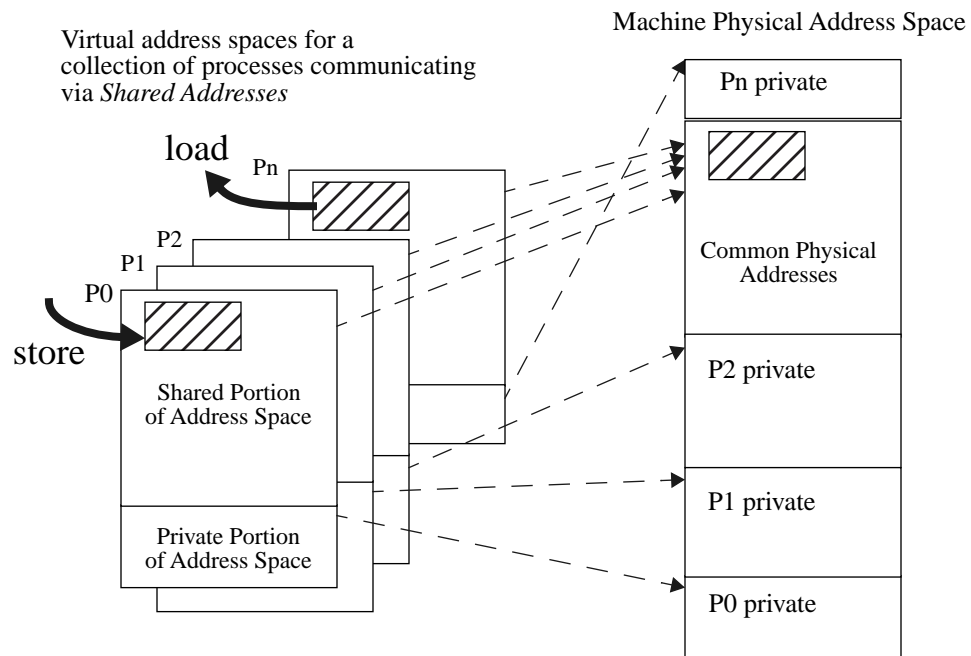
Figure 1-14 Typical memory model for shared-memory parallel programs

Collection of processes have a common region of physical addresses mapped into their virtual address space, in addition to the private region, which typically contains the stack and private data.

devices to I/O controllers or by inserting additional I/O controllers. There are two possible ways to increase the processing capacity: wait for a faster processor to become available, or add more processors. On a timesharing workload, this should increase the throughput of the system. With more processors, more processes run at once and throughput is increased. If a single application is programmed to make use of multiple threads, more processors should speed up the application. The primitives provided by the hardware are essentially one-to-one with the operations available in the programming model.

Within the general framework of Figure 1-15, there has been a great deal of evolution of shared memory machines as the underlying technology advanced. The early machines were "high end" mainframe configurations [Lon61,Padeg]. On the technology side, memory in early mainframes was slow compared to the processor, so it was necessary to interleave data across several memory banks to obtain adequate bandwidth for a single processor; this required an interconnect between the processor and each of the banks. On the application side, these systems were primarily designed for throughput on a large number of jobs. Thus, to meet the I/O demands of a workload, several I/O channels and devices were attached. The I/O channels also required direct access each of the memory banks. Therefore, these systems were typically organized with a *cross-bar switch* connecting the CPU and several I/O channels to several memory banks, as indicated by Figure 1-16a. Adding processors was primarily a matter of expanding the switch; the hardware structure to access a memory location from a port on the processor and I/O side of the switch was unchanged. The size and cost of the processor limited these early systems to a small number of processors. As the hardware density and cost improved, larger systems could be contemplated. The cost of scaling the cross-bar became the limiting factor, and in many cases it was replaced by a *multi-*
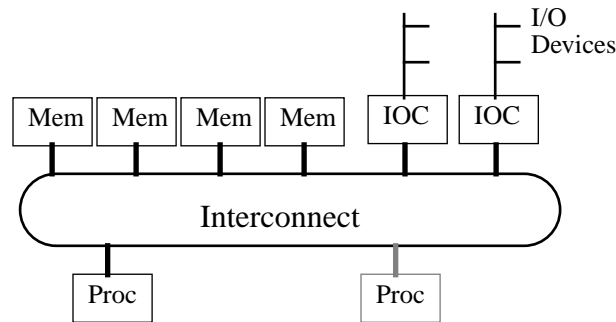
Figure 1-15 Extending a system into a shared-memory multiprocessor by adding processor modules

Most systems consist of one or more memory modules accessible by a processor and I/O controllers through a hardware interconnect, typically a bus, cross-bar or multistage interconnect. Memory and I/O capacity is increased by attaching memory and I/O modules. Shared-memory machines allow processing capacity to be increased by adding processor modules.

stage interconnect, suggested by Figure 1-16b, for which the cost increases more slowly with the number of ports. These savings come at the expense of increased latency and decreased bandwidth per port, if all are used at once. The ability to access all memory directly from each processor has several advantages: any processor can run any process or handle any I/O event and within the operating system data structures can be shared.

(a) Cross-bar Switch   (b) Multistage Interconnection Network   (c) Bus Interconnect
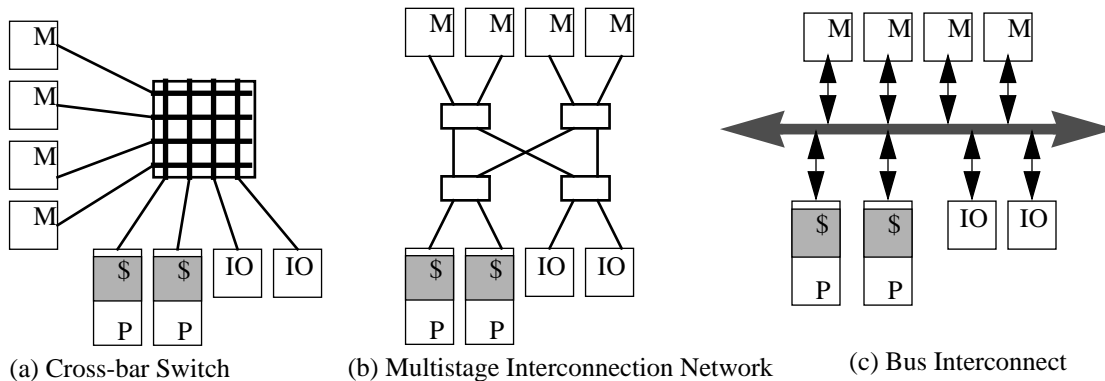
Figure 1-16 Typical Shared-Memory Multiprocessor Interconnection schemes.

The interconnection of multiple processors, with their local caches (indicated by $), and I/O controllers to multiple memory modules may be a cross-bar, multistage interconnection network, or bus.

The widespread use of shared-memory multiprocessor designs came about with the 32-bit microprocessor revolution in the mid 80s, because the processor, cache, floating point, and memory management unit fit on a single board[Bel95], or even two to a board. Most mid-range machines, including minicomputers, servers, workstations, and personal computers are organized around a central memory bus, as illustrated in Figure 1-16a. The standard bus access mechanism allows any processor to access any physical address in the system. Like the switch based designs, all memory locations are equidistant to all processors, so all processors experience the same access time, or latency, on a memory reference. This configuration is usually called a *symmetric multiprocessor* (SMP)[1]. SMPs are heavily used for execution of parallel programs, as well as multi-

programming. The typical organization of bus-based symmetric multiprocessor is illustrated by Figure 1-17, which describes the first highly integrated SMP for the commodity market. Figure 1-18 illustrates a high-end server organization which distributes the physical memory over the processor modules.

The factors limiting the number of processors that can be supported with a bus-based organization are quite different from those in the switch-based approach. Adding processors to the switch is expensive, however, the aggregate bandwidth increases with the number of ports. The cost of adding a processor to the bus is small, but the aggregate bandwidth is fixed. Dividing this fixed bandwidth among the larger number of processors limits the practical scalability of the approach. Fortunately, caches reduce the bandwidth demand of each processor, since many references are satisfied by the cache, rather than by the memory. However, with data replicated in local caches there is a potentially challenging problem of keeping the caches "consistent", which we will examine in detail later in the book.

Starting from a baseline of small-scale shared memory machines, illustrated in the figures above, we may ask what is required to scale the design to a large number of processors. The basic processor component is well-suited to the task, since it is small and economical, but there is clearly a problem with the interconnect. The bus does not scale because it has a fixed aggregate bandwidth. The cross-bar does not scale well, because the cost increases as the square of the number of ports. Many alternative scalable interconnection networks exist, such that the aggregate bandwidth increases as more processors are added, but the cost does not become excessive. We need to be careful about the resulting increase in latency, because the processor may stall while a memory operation moves from the processor to the memory module and back. If the latency of access becomes too large, the processors will spend much of their time waiting and the advantages of more processors may be offset by poor utilization.

One natural approach to building scalable shared memory machines is to maintain the uniform memory access (or "dancehall") approach of Figure 1-15 and provide a scalable interconnect between the processors and the memories. Every memory access is translated into a message transaction over the network, much as it might be translated to a bus transaction in the SMP designs. The primary disadvantage of this approach is that the round-trip network latency is experienced on every memory access and a large bandwidth must be supplied to every processor.

An alternative approach is to interconnect complete processors, each with a local memory, as illustrated in Figure 1-19. In this non-uniform memory access (NUMA) approach, the local memory controller determines whether to perform a local memory access or a message transaction with a remote memory controller. Accessing local memory is faster than accessing remote memory. (The I/O system may either be a part of every node or consolidated into special I/O nodes, not shown.) Accesses to private data, such as code and stack, can often be performed locally, as can accesses to shared data that, by accident or intent, are stored on the local node. The ability to access the local memory quickly does not increase the time to access remote data appreciably, so it reduces the average access time, especially when a large fraction of the accesses are

---

1. The term SMP is widely used, but causes a bit of confusion. What exactly needs to be symmetric? Many designs are symmetric in some respect. The more precise description of what is intended by SMP is a shared memory multiprocessor where the cost of accessing a memory location is the same for all processors, i.e., it has *uniform* access costs when the access actually is to memory. If the location is cached, the access will be faster, but still it is symmetric with respect to processors.
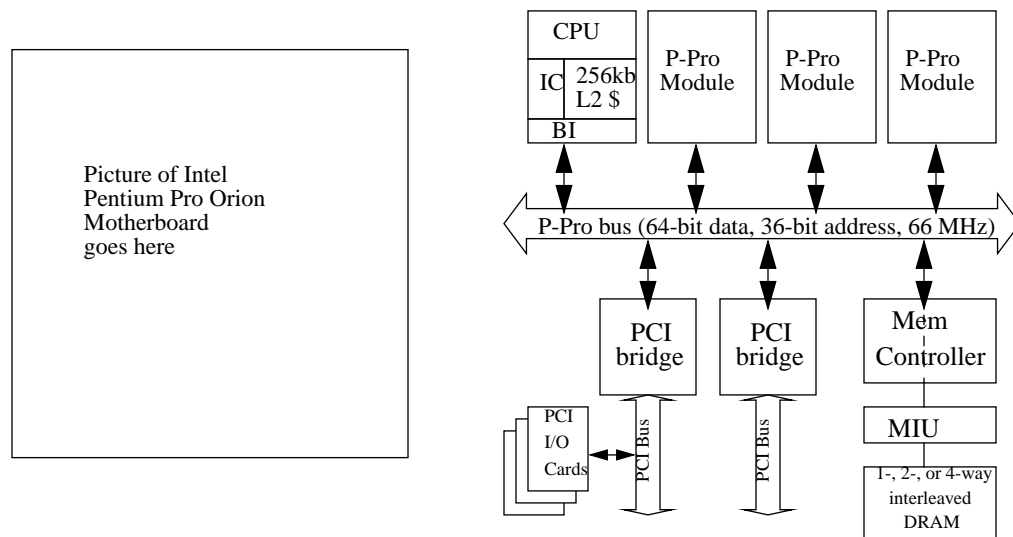
Figure 1-17 Physical and logical organization of the Intel PentiumPro four processor "quad pack"

The Intel quad-processor Pentium Pro motherboard employed in many multiprocessor servers illustrates the major design elements of most small scale SMPs. Its logical block diagram shows there can be up to four processor modules, each containing a Pentium-Pro processor, first level caches, TLB, 256 KB second level cache, interrupt controller (IC), and a bus interface (BI) in a single chip connecting directly to a 64-bit memory bus. The bus operates at 66 MHz and memory transactions are pipelined to give a peak bandwidth of 528 MB/s. A two-chip memory controller and four-chip memory interleave unit (MIU) connect the bus to multiple banks of DRAM. Bridges connect the memory bus to two independent PCI busses, which host display, network, SCSI, and lower speed I/O connections. The Pentium-Pro module contains all the logic necessary to support the multiprocessor communication architecture, including that required for memory and cache consistency. The structure of the Pentium-Pro "quad pack" is similar to a large number of earlier SMP designs, but has a much higher degree of integration and is targeted at a much larger volume.

to local data. The bandwidth demand placed on the network is also reduced. Although some conceptual simplicity arises from having all shared data equidistant from any processor, the NUMA approach has become far more prevalent because of its inherent performance advantages and because it harnesses more of the mainstream processor memory system technology. One example of this style of design is the Cray T3E, illustrated in Figure 1-20. This machine reflects the viewpoint where, although all memory is accessible to every processor, the distribution of memory across processors is exposed to the programmer. Caches are used only to hold data (and instructions) from local mmory. It is the programmer's job to avoid frequent report references. The SGI Origin is an example of a machine with a similar organizational structure, but it allows data from any memory to be replicated into any of the caches and provide hardware support to keep these caches consistent, without relying on a bus connecting all the modules with a common set of wires. While this book was being written, the two dsigns literally converged after the two companies merged.

To summarize, the communication abstraction underlying the shared address space programming model is reads and writes to shared variables, this is mapped one-to-one to a communication abstraction consisting of load and store instructions accessing a global, shared address space, which is supported directly in hardware through access to shared physical memory locations. The communication abstraction is very "close" to the actual hardware. Each processor can *name*
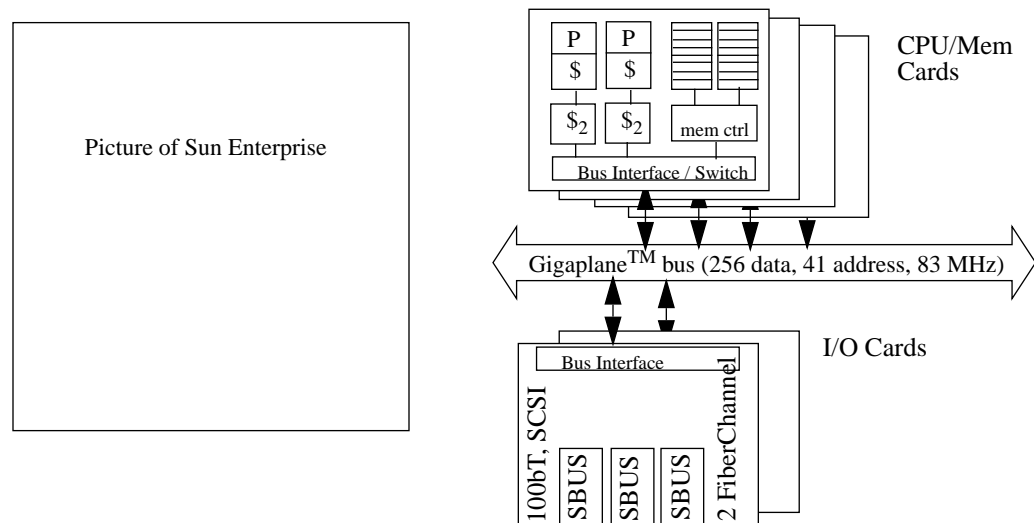
Figure 1-18 Physical and logical organization of the Sun Enterprise Server

A larger scale design is illustrated by the Sun Ultrasparc-based Enterprise multiprocessor server. The diagram shows its physical structure and logical organization. A wide (256 bit), highly pipelined memory bus delivers 2.5 GB/s of memory bandwidth. This design uses a hierarchical structure, where each card is either a complete dual-processor with memory or a complete I/O system. The full configuration supports 16 cards of either type, with at least one of each. The CPU/Mem card contains two Ultrasparc processor modules, each with 16 KB level-1 and 512 KB level-2 caches, plus two 512-bit wide memory banks and an internal switch. Thus, adding processors adds memory capacity and memory interleaving. The I/O card provides three SBUS slots for I/O extensions, a SCSI connector, 100bT Ethernet port, and two FiberChannel interfaces. A typical complete configuration would be 24 processors and 6 I/O cards. Although memory banks are physically packaged with pairs of processors, all memory is equidistant from all processors and accessed over the common bus, preserving the SMP characteristics. Data may be placed anywhere in the machine with no performance impact.
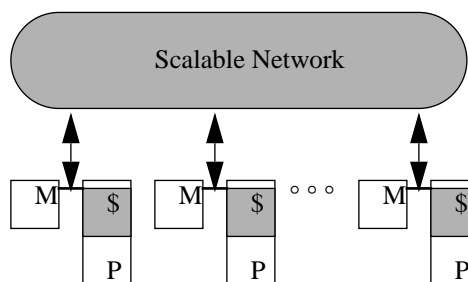
Figure 1-19 Non-uniform Memory Access (NUMA) Scalable shared memory multiprocessor organization.

Processor and memory modules are closely integrated such that access to local memory is faster than access to remote memories.

every physical location in the machine; a process can name all data it shares with others within its virtual address space. Data transfer is a result of conventional load and store instructions, and the data is transferred either as primitive types in the instruction set, bytes, words, etc., or as cache blocks. Each process performs memory operations on addresses in its virtual address space; the
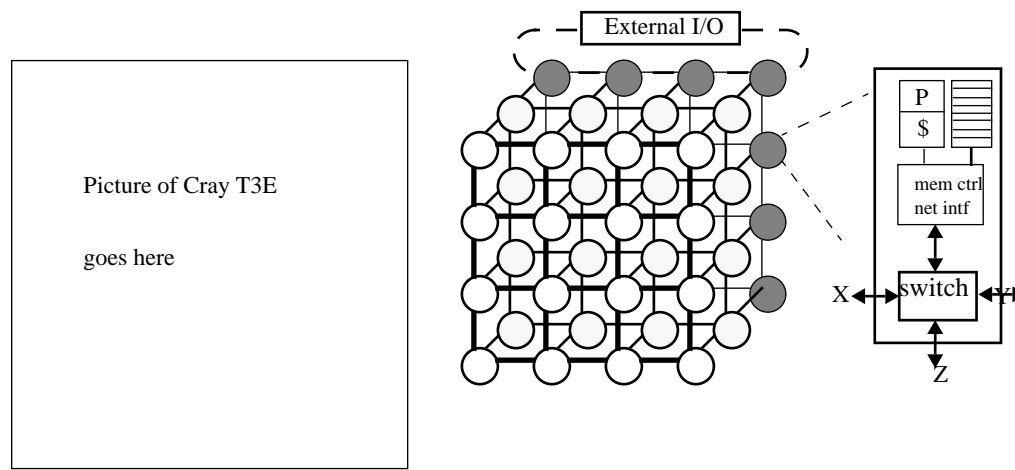
Figure 1-20 Cray T3E Scalable Shared Address Space Machine

The Cray T3E is designed to scale up to a thousand processors supporting a global shared address space. Each node contains a DEC Alpha processor, local memory, a network interface integrated with the memory controller, and a network switch. The machine is organized as a three dimensional cube, with each node connected to its six neighbors through 480 MB/s point-to-point links. Any processor can read or write any memory location, however, the NUMA characteristic of the machine is exposed in the communication architecture, as well as in its performance characteristics. A short sequence of instructions is required to establish addressability to remote memory, which can then be accessed by conventional loads and stores. The memory controller captures the access to a remote memory and conducts a message transaction with the memory controller of the remote node on the local processor's behalf. The message transaction is automatically routed through intermediate nodes to the desired destination, with a small delay per "hop". The remote data is typically not cached, since there is no hardware mechanism to keep it consistent. (We will look at other design points that allow shared data to be replicated throughout the processor caches.) The Cray T3E I/O system is distributed over a collection of nodes on the surface of the cube, which are connected to the external world through an addition I/O network

address translation process identifies a physical location, which may be local or remote to the processor and may be shared with other processes. In either case, the hardware accesses it directly, without user or operating system software intervention. The address translation realizes protection within the shared address space, just as it does for uniprocessors, since a process can only access the data in its virtual address space. The effectiveness of the shared memory approach depends on the latency incurred on memory accesses, as well as the bandwidth of data transfer that can be supported. Just as a memory storage hierarchy allows that data that is bound to an address to be migrated toward the processor, expressing communication in terms of the storage address space allows shared data to be migrated toward the processor that accesses it. However, migrating and replicating data across a general purpose interconnect presents a unique set of challenges. We will see that to achieve scalability in such a design it is necessary, but not sufficient, for the hardware interconnect to scale well. The entire solution, including the mechanisms used for maintaining the consistent shared memory abstractions, must also scale well.

1.3.3 Message-Passing

A second important class of parallel machines, *message passing architectures*, employs complete computers as building blocks – including the microprocessor, memory and I/O system – and provides communication between processors as explicit I/O operations. The high-level block dia-

gram for a message passing machine is essentially the same as the NUMA shared-memory approach, shown in Figure 1-19. The primary difference is that communication is integrated at the I/O level, rather than into the memory system. This style of design also has much in common with networks of workstations, or "clusters", except the packaging of the nodes is typically much tighter, there is no monitor or direct user access, and the network is of much higher capability than standard local area network. The integration between the processor and the network tends to be much tighter than in traditional I/O structures, which support connection to devices that are much slower than the processor, since message passing is fundamentally processor-to-processor communication.

In message-passing there is generally a substantial distance between the programming model and the communication operations at the physical hardware level, because user communication is performed through operating system or library calls which perform many lower level actions, including the actual communication operation. Thus, the discussion of message-passing begins with a look at the communication abstraction, and then briefly surveys the evolution of hardware organizations supporting this abstraction.

The most common user-level communication operations on message passing systems are variants of *send* and *receive*. In its simplest form, send specifies a local data buffer that is to be transmitted and a receiving process (typically on a remote processor). Receive specifies a sending process and a local data buffer into which the transmitted data is to be placed. Together, the matching send and receive cause a data transfer from one process to another, as indicated in Figure 1-19. In most message passing systems, the send operation also allows an identifier or *tag* to be attached to the message and the receiving operation specifies a matching rule, such as a specific tag from a specific processor, any tag from any processor. Thus, the user program names local addresses and entries in an abstract process-tag space. *The combination of a send and a matching receive accomplishes a memory to memory copy, where each end specifies its local data address, and a pairwise synchronization event.* There are several possible variants of this synchronization event, depending upon whether the send completes when the receive has been executed, when the send buffer is available for reuse, or when the request has been accepted. Similarly, the receive can potentially wait until a matching send occurs or simply post the receive. Each of these variants have somewhat different semantics and different implementation requirements.

Message passing has long been used as a means of communication and synchronization among arbitrary collections of cooperating sequential processes, even on a single processor. Important examples include programming languages, such as CSP and Occam, and common operating systems functions, such as sockets. Parallel programs using message passing are typically quite structured, like their shared-memory counter parts. Most often, all nodes execute identical copies of a program, with the same code and private variables. Usually, processes can name each other using a simple linear ordering of the processes comprising a program.

Early message passing machines provided hardware primitives that were very close to the simple send/receive user-level communication abstraction, with some additional restrictions. A node was connected to a fixed set of neighbors in a regular pattern by point-to-point links that behaved as simple FIFOs[Sei85]. This sort of design is illustrated in Figure 1-22 for a small 3D cube. Most early machines were *hypercubes*, where each node is connected to n other nodes differing by one bit in the binary address, for a total of 2^n nodes, or *meshes*, where the nodes are connect to neighbors on two or three dimensions. The network topology was especially important in the early message passing machines, because only the neighboring processors could be named in a send or receive operation. The data transfer involved the sender writing into a link and the
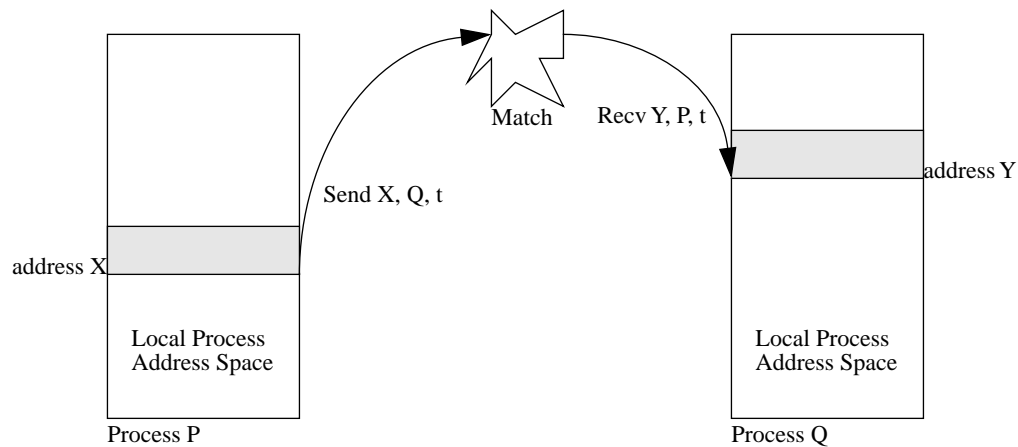
Figure 1-21 User level send/receive message passing abstraction

A data transfer from one local address space to another occurs when a send to a particular process is matched with a received postd by that process.

receiver reading from the link. The FIFOs were small, so the sender would not be able to finish writing the message until the receiver started reading it, so the send would block until the receive occured. In modern terms this is called *synchronous* message passing because the two events coincide in time. The details of moving data were hidden from the programmer in a message passing library, forming a layer of software between send and receive calls and the actual hardware.[1]

The direct FIFO design was soon replaced by more versatile and more robust designs which provided direct memory access (DMA) transfers on either end of the communication event. The use of DMA allowed *non-blocking* sends, where the sender is able to initiate a send and continue with useful computation (or even perform a receive) while the send completes. On the receiving end, the transfer is accepted via a DMA transfer by the message layer into a buffer and queued until the target process performs a matching receive, at which point the data is copying into the address space of the receiving process.

The physical topology of the communication network dominated the programming model of these early machines and parallel algorithms were often stated in terms of a specific interconnection topology, e.g., a ring, a grid, or a hypercube[Fox*88]. However, to make the machines more generally useful, the designers of the message layers provided support for communication between arbitrary processors, rather than only between physical neighbors[NX]. This was originally supported by forwarding the data within the message layer along links in the network. Soon this routing function was moved into the hardware, so each node consisted of a processor with mmory, and a switch that could forward messsages, called a router. However, in this *store-and-forward* approach the time to transfer a message is proportional to the number of hops it takes

---

1. The motivation for synchronous message passing was not just from the machine structure; it was also was also present in important programming languages, especially CSP[***CSP]. Early in the microprocesor era the approach was captured in a single chip building block, the Transputer, which was widely touted during its development by INMOS as a revolution in computing.
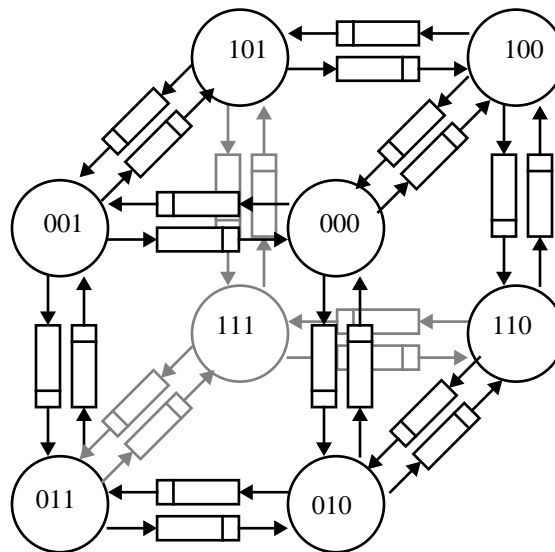
Figure 1-22 Typical structure of an early message passing machines

Each node is connected to neighbors in three dimensions via FIFOs.

through the network, so there remained an emphasis on interconnection topology.(See Exercise 1.7)

The emphasis on network topology was significantly reduced with the introduction of more general purpose networks, which pipelined the message transfer through each of the routers forming the interconnection network[Bar*94,BoRo89,Dun88,HoMc93,Lei*92,PiRe94,VEi*92]. In most modern message passing machines, the incremental delay introduced by each router is small enough that the transfer time is dominated by the time to simply move that data between the processor and the network, not how far it travels.[Gro92,HoMc93,Hor*93PiRe94]. This greatly simplifies the programming model; typically the processors are viewed as simply forming a linear sequence with uniform communication costs. In other words, the communication abstraction reflects an organizational structure much as in Figure 1-19. One important example of such machine is the IBM SP-2, illustrated in Figure 1-23, which is constructed from conventional RS6000 workstations, a scalable network, and a network interface containing a dedicated processor. Another is the Intel Paragon, illustrated in Figure 1-24, which integrates the network interface more tightly to the processors in an SMP nodes, where one of the processors is dedicated to supporting message passing.

A processor in a message passing machine can name only the locations in its local memory, and it can name each of the procesors, perhaps by number or by route. A user process can only name private addresses and other processes; it can transfer data using the send/receive calls.

1.3.4 Convergence

Evolution of the hardware and software has blurred the once clear boundary between the shared memory and message passing camps. First, consider the communication operations available to the user process.
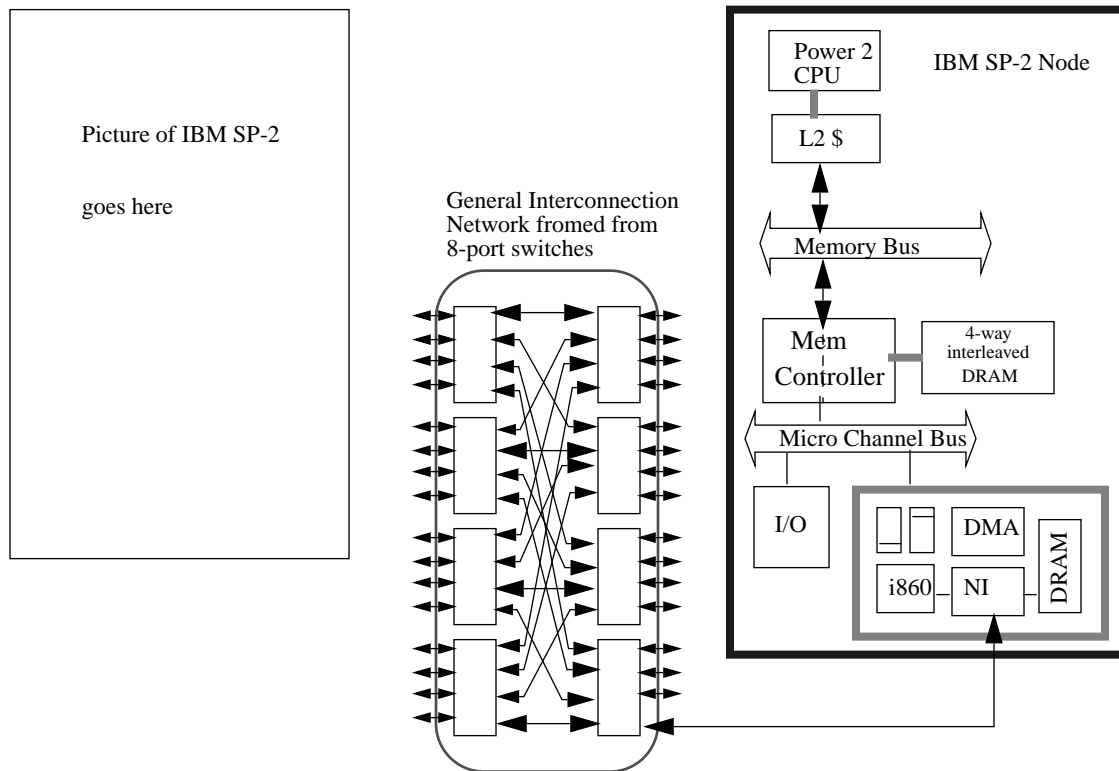
Figure 1-23   IBM SP-2 Message Passing Machine

The IBM SP-2 is a scalable parallel machine constructed essentially out of complete RS6000 workstations. Modest modifications are made to packaging the workstations into standing racks. A network interface card (NIC) is inserted at the MicroChannel I/O bus. The NIC contains the drivers for the actual link into the network, a substantial amount of memory to buffer message data, and a complete i960 microprocessor to move data between host memory and the network. The network itself is a butterfly-like structure, constructed by cascading 8x8 cross-bar switches. The links operate at 40 MB/s in each direction, which is the full capability of the I/O bus. Several other machine employ a similar network interface design, but connect directly to the memory bus, rather than at the I/O bus.

- Traditional message passing operations (send/receive) are supported on most shared memory machines through shared buffer storage. Send involves writing data, or a pointer to data, into the buffer, receive involves reading the data from shared storage. Flags or locks are used to control access to the buffer and to indicate events, such as message arrival.

- On a message passing machine, a user process may construct a global address space of sorts by carrying along pointers specifying the process and local virtual address in that process. Access to such a global address can be performed in software, through an explicit message transaction. Most message passing libraries allow a process to accept a message for "any" process, so each process can serve data requests from the others. A logical read is realized by sending an request to the process containing the object and receiving a response. The actual message transaction may be hidden from the user; it may be carried out by compiler generated code for access to a shared variable.[Split-C,Midway,CID,Cilk]
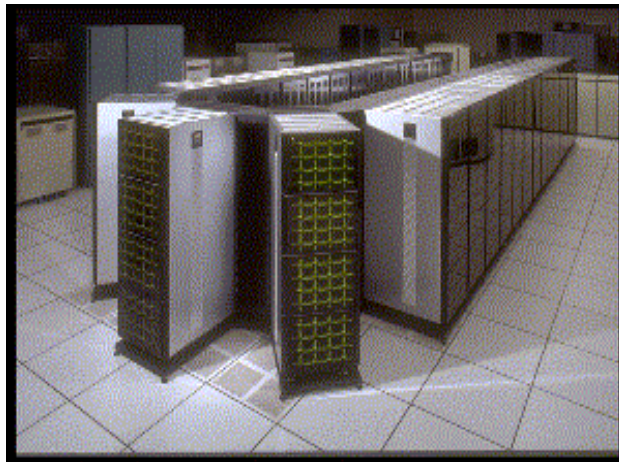
Source: http://www.cs.sandia.gov/gif/paragon.gif
courtesy of Sandia National Laboratory
1824 nodes configured as a 16
high by 114 wide array
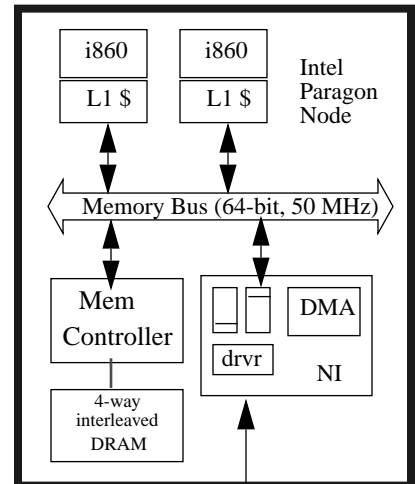
2D grid network
with processing node
attached to every switch

8 bits
175 MHz
bidirectional
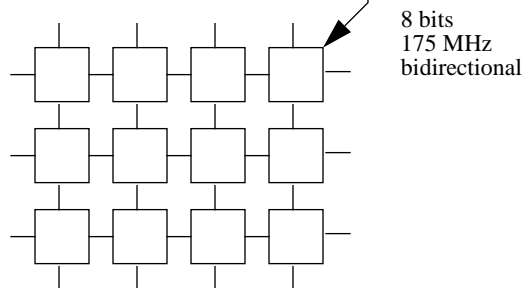
Figure 1-24   Intel Paragon

The Intel Paragon illustrates a much tighter packaging of nodes. Each card is an SMP with two or more i860 processors and a network interface chip connected to the cache-coherent memory bus. One of the processors is dedicated to servicing the network. In addition, the node has a DMA engine to transfer contiguous chunks of data to and from the network at a high rate. The network is a 3D grid, much like the Cray T3E, with links operating at 175 MB/s in each direction.

- A shared virtual address space can be established on a message passing machine at the page level. A collection of processes have a region of shared addresses, but for each process only the pages that are local to it are accessible. Upon access to a missing (i.e., remote) page, a page fault occurs and the operating system engages the remote node in a message transaction to transfer the page and map it into the user address space.

At the level of machine organization, there has been substantial convergence as well. Modern message passing architectures appear essentially identical at the block diagram level to the scalable NUMA design illustrated in Figure 1-19. In the shared memory case, the network interface was integrated with the cache controller or memory controller, in order for that device to observe cache misses and conduct a message transaction to access memory in a remote node. In the message passing approach, the network interface is essentially an I/O device. However, the trend has been to integrate this device more deeply into the memory system as well, and to transfer data directly from the user address space. Some designs provide DMA transfers across the network, from memory on one machine to memory on the other machine, so the network interface is inte-

grated fairly deeply with the memory system. Message passing is implemented on top of these remote memory copies[HoMc92]. In some designs a complete processor assists in communication, sharing a cache-coherent memory bus with the main processor[Gro92,PiRe94,HoMc92]. Viewing the convergence from the other side, clearly all large-scale shared memory operations are ultimately implemented as message transactions at some level.

In addition to the convergeence of scalable message passing and shared memory machines, switched-based local area networks, including fast ethernet, ATM, FiberChannel, and several proprietary design[Bod95, Gil96] have emerged, providing scalable interconnects that are approaching what traditional parallel machines offer. These new networks are being used to connect collections of machines, which may be shared-memory multiprocessors in their own right, into *clusters*, which may operate as a parallel machine on individual large problems or as many individual machines on a multiprogramming load. Also, essentially all SMP vendors provide some form of network clustering to obtain better reliability.

In summary, message passing and a shared address space represent two clearly distinct programming models, each providing a well-defined paradigm for sharing, communication, and synchronization. However, the underlying machine structures have converged toward a common organization, represented by a collection of complete computers, augmented by a *communication assist* connecting each node to a scalable communication network. Thus, it is natural to consider supporting aspects of both in a common framework. Integrating the communication assist more tightly into the memory system tends to reduce the latency of network transactions and improve the bandwidth that can be supplied to or accepted from the network. We will want to look much more carefully at the precise nature of this integration and understand how it interacts with cache design, address translation, protection, and other traditional aspects of computer architecture.

1.3.5 Data Parallel Processing

A third important class of parallel machines has been variously called: processor arrays, single-instruction-multiple-data machines, and data parallel architectures. The changing names reflect a gradual separation of the user-level abstraction from the machine operation. *The key characteristic of the programming model is that operations can be performed in parallel on each element of a large regular data structure, such as an array or matrix*. The program is logically a single thread of control, carrying out a sequence of either sequential or parallel steps. Within this general paradigm, there has been many novel designs, exploiting various technological opportunities, and considerable evolution as microprocessor technology has become such a dominate force.

An influential paper in the early 70's[Fly72] developed a taxonomy of computers, known as *Flynn's taxonomy* , which characterizes designs in terms of the number of distinct instructions issued at a time and the number of data elements they operate on. Conventional sequential computers being single-instruction-single-data (SISD) and parallel machines built from multiple conventional processors being multiple-instruction-multiple-data (MIMD). The revolutionary alternative was single-instruction-multiple-data (SIMD). Its history is rooted in the mid-60s when an individual processor was a cabinet full of equipment and an instruction fetch cost as much in time and hardware as performing the actual instruction. The idea was that all the instruction sequencing could be consolidated in the control processor. The data processors included only the ALU, memory, and a simple connection to nearest neighbors.

In the early data parallel machines, the data parallel programming model was rendered directly in the physical hardware[Bal*62, Bou*72,Cor72,Red73,Slo*62,Slot67,ViCo78]. Typically, a control processor broadcast each instruction to an array of data processing elements (PEs), which were connected to form a regular grid, as suggested by Figure 1-25. It was observed that many important scientific computations involved uniform calculation on every element of an array or matrix, often involving neighboring elements in the row or column. Thus, the parallel problem data was distributed over the memories of the data processors and scalar data was retained in the control processor's memory. The control processor instructed the data processors to each perform an operation on local data elements or all to perform a communication operation. For example, to average each element of a matrix with its four neighbors, a copy of the matrix would be shifted across the PEs in each of the four directions and a local accumulation performed in each PE. Data PEs typically included a condition flag, allowing some to abstain from an operation. In some designs, the local address can be specified with an indirect addressing mode, allowing processors to all do the same operation, but with different local data addresses.
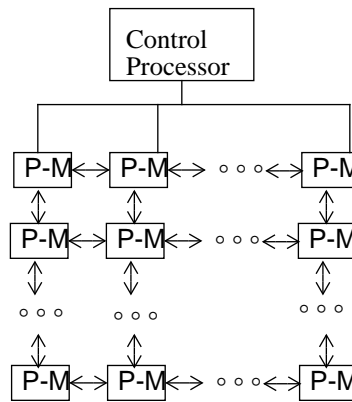
Figure 1-25 Typical organization of a data parallel (SIMD) parallel machine

Individual processing elements (PEs) operate in lock-step under the direction of a single control processor. Traditionally, SIMD machines provide a limited, regular interconnect among the PEs, although this was generalized in later machines, such as the Thinking Machines Connection Machine and the MasPar.

The development of arrays of processors was almost completely eclipsed in the mid 70s with the development of *vector processors*. In these machines, a scalar processor is integrated with a collection of function units that operate on vectors of data out of one memory in a pipelined fashion. The ability to operate on vectors anywhere in memory eliminated the need to map application data structures onto a rigid interconnection structure and greatly simplified the problem of getting data aligned so that local operations could be performed. The first vector processor, the CDC Star-100, provided vector operations in its instruction set that combined two source vectors from memory and produced a result vector in memory. The machine only operated at full speed if the vectors were contiguous and hence a large fraction of the execution time was spent simply transposing matrices. A dramatic change occurred in 1976 with the introduction of the Cray-1, which extended the concept of a load-store architecture employed in the CDC 6600 and CDC 7600 (and rediscovered in modern RISC machines) to apply to vectors. Vectors in memory, of any fixed stride, were transferred to or from contiguous vector registers by vector load and store instructions. Arithmetic was performed on the vector registers. The use of a very fast scalar processor (operating at the unprecedented rate of 80 MHz) tightly integrated with the vector operations and

utilizing a large semiconductor memory, rather than core, took over the world of supercomputing. Over the next twenty years Cray Research led the supercomputing market by increasing the bandwidth for vector memory transfers, increasing the number of processors, the number of vector pipelines, and the length of the vector registers, resulting in the performance growth indicated by Figure 1-10 and Figure 1-11.

The SIMD data parallel machine experienced a renaissance in the mid-80s, as VLSI advances made simple 32-bit processor just barely practical[Bat79,Bat80,Hill85,Nic90,TuRo88]. The unique twist in the data parallel regime was to place thirty-two very simple 1-bit processing elements on each chip, along with serial connections to neighboring processors, while consolidating the instruction sequencing capability in the control processor. In this way, systems with several thousand bit-serial processing elements could be constructed at reasonable cost. In addition, it was recognized that the utility of such a system could be increased dramatically with the provision of a general interconnect allowing an arbitrary communication pattern to take place in a single rather long step, in addition to the regular grid neighbor connections[Hill85,HiSt86,Nic90]. The sequencing mechanism which expands conventional integer and floating point operations into a sequence of bit serial operations also provided a means of "virtualizing" the processing elements, so that a few thousand processing elements can give the illusion of operating in parallel on millions of data elements with one virtual PE per data element.

The technological factors that made this bit-serial design attractive also provided fast, inexpensive, single-chip floating point units, and rapidly gave way to very fast microprocessors with integrated floating point and caches. This eliminated the cost advantage of consolidating the sequencing logic and provided equal peak performance on a much smaller number of complete processors. The simple, regular calculations on large matrices which motivated the data parallel approach also have tremendous spatial and temporal locality, if the computation is properly mapped onto a smaller number of complete processors, with each processor responsible for a large number of logically contiguous data points. Caches and local memory can be brought to bear on the set of data points local to each node, while communication occurs across the boundaries or as a global rearrangement of data.

Thus, while the user-level abstraction of parallel operations on large regular data structures continued to offer an attractive solution to an important class of problems, the machine organization employed with data parallel programming models evolved towards a more generic parallel architecture of multiple cooperating microprocessors, much like scalable shared memory and message passing machines, often with the inclusion of specialized network support for global synchronization, such as a *barrier*, which causes each process to wait at a particular point in the program until all other processes have reach that point[Hor93,Lei*92,Kum92,KeSc93,Koe*94]. Indeed, the SIMD approach evolved into the SPMD (single-program-multiple-data) approach, in which all processors execute copies of the same program, and has thus largely converged with the more structured forms of shared memory and message passing programming.

Data parallel programming languages are usually implemented by viewing the local address spaces of a collection of processes, one per processor, as forming an explicit global address space. Data structures are laid out across this global address space and there is a simple mapping from indexes to processor and local offset. The computation is organized as a sequence of "bulk synchronous" phases of either local computation or global communication, separated by a global barrier [cite BSP]. Because all processors do communication together and there is a global view of what is going on, either a shared address space or message passing can be employed. For example, if a phase involved every processor doing a write to an address in the processor "to the

left", it could be realized by each doing a send to the left and a receive "from the right" into the destination address. Similarly, every processor reading can be realized by every processor sending the address and then every processor sending back the data. In fact, the code that is produced by compilers for modern data parallel languages is essentially the same as for the structured control-parallel programs that are most common in shared-memory and message passing programming models. With the convergence in machine structure, there has been a convergence in how the machines are actually used.

1.3.6 Other Parallel Architectures

The mid-80s renaissance gave rise to several other architectural directions which received considerable investigation by academia and industry, but enjoyed less commercial success than the three classes discussed above and therefore experienced less use as a vehicle for parallel programming. Two approaches that were developed into complete programming systems were dataflow architectures and systolic architectures. Both represent important conceptual developments of continuing value as the field evolves.

Dataflow Architecture

Dataflow models of computation sought to make the essential aspects of a parallel computation explicit at the machine level, without imposing artificial constraints that would limit the available parallelism in the program. The idea is that the program is represented by a graph of essential data dependences, as illustrated in Figure 1-26, rather than as a fixed collection of explicitly sequenced threads of control. An instruction may execute whenever its data operands are available. The graph may be spread arbitrarily over a collection of processors. Each node specifies an operation to perform and the address of each of the nodes that need the result. In the original form, a processor in a dataflow machine operates a simple circular pipeline. A message, or *token*, from the network consists of data and an address, or *tag*, of its destination node. The tag is compared against those in a matching store. If present, the matching token is extracted and the instruction is issued for execution. If not, the token is placed in the store to await its partner. When a result is computed, a new message, or token, containing the result data is sent to each of the destinations specified in the instruction. The same mechanism can be used whether the successor instructions are local or on some remote processor. The primary division within dataflow architectures is whether the graph is *static*, with each node representing a primitive operation, or *dynamic*, in which case a node can represent the invocation of an arbitrary function, itself represented by a graph. In dynamic or *tagged-token* architectures, the effect of dynamically expanding the graph on function invocation is usually achieved by carrying additional context information in the tag, rather than actually modifying the program graph.

The key characteristic of dataflow architectures is the ability to name operations performed anywhere in the machine, the support for synchronization of independent operations, and dynamic scheduling at the machine level. As the dataflow machine designs matured into real systems, programmed in high level parallel languages, a more conventional structure emerged. Typically, parallelism was generated in the program as a result of parallel function calls and parallel loops, so it was attractive to allocate these larger chunks of work to processors. This led to a family of designs organized essentially like the NUMA design of Figure 1-19. The key differentiating features being direct support for a large, dynamic set of threads of control and the integration of communication with thread generation. The network was closely integrated with the processor; in many designs the "current message" is available in special registers, and there is hardware sup-

a=(b+1)*(b-c)
d=c*e
f=a*d
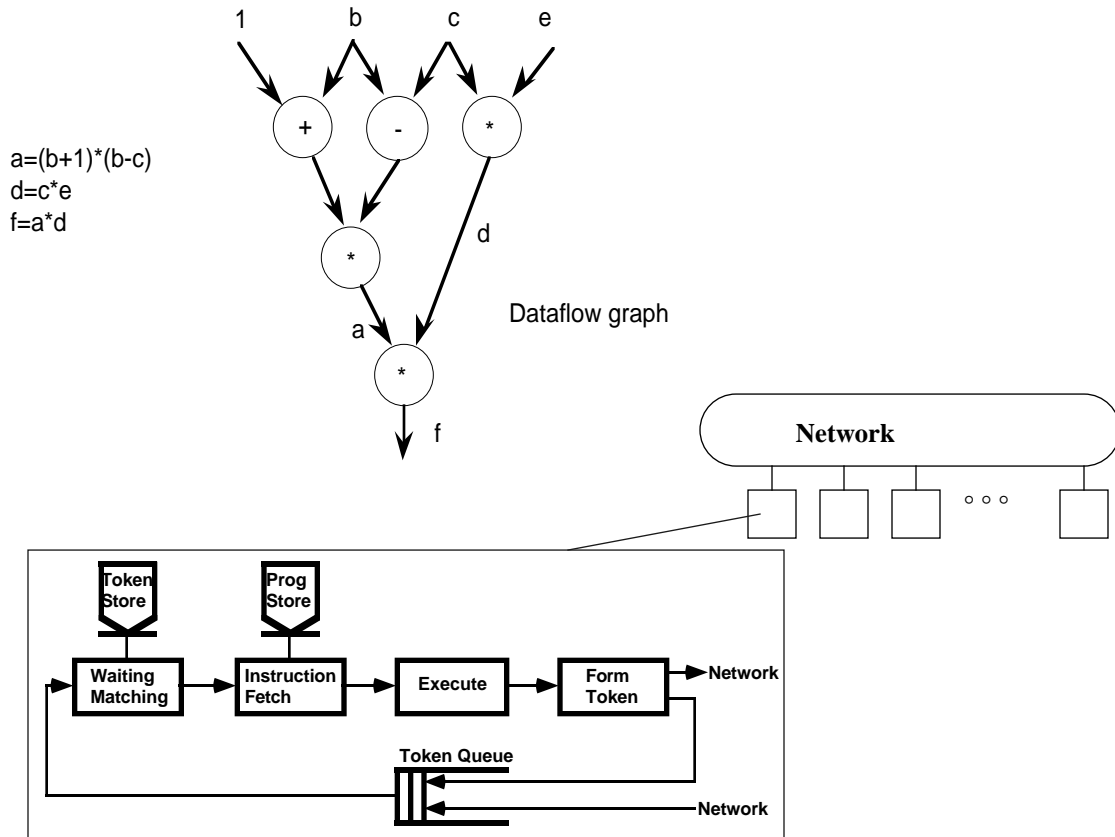
Figure 1-26 Dataflow graph and basic execution pipeline

port for dispatching to a thread identified in the message. In addition, many designs provide extra state bits on memory locations in order to provide fine-grained synchronization, i.e., synchronization on an element-by-element basis, rather than using locks to synchronize accesses to an entire data structure. In particular, each message could schedule a chunk of computation which could make use of local registers and memory.

By contrast, in shared memory machines one generally adopts the view that a static or slowly varying set of processes operate within a shared address space, so the compiler or program maps the logical parallelism in the program to a set of processes by assigning loop iterations, maintaining a shared work queue, or the like. Similarly, message passing programs involve a static, or nearly static, collection of processes which can name one another in order to communicate. In data parallel architectures, the compiler or sequencer maps a large set of "virtual processor" operations onto processors by assigning iterations of a regular loop nest. In the dataflow case, the machine provides the ability to name a very large and dynamic set of threads which can be mapped arbitrarily to processors. Typically, these machines provided a global address space as well. As we have seen with message passing and data parallel machines, dataflow architectures experienced a gradual separation of programming model and hardware structure as the approach matured.

Systolic Architectures

Another novel approach was *systolic architectures*, which sought to replace a single sequential processor by a regular array of simple processing elements and, by carefully orchestrating the flow of data between PEs, obtain very high throughput with modest memory bandwidth requirements. These designs differ from conventional pipelined function units, in that the array structure can be non-linear, *e.g.* hexagonal, the pathways between PEs may be multidirectional, and each PE may have a small amount of local instruction and data memory. They differ from SIMD in that each PE might do a different operation.

The early proposals were driven by the opportunity offered by VLSI to provide inexpensive special purpose chips. A given algorithm could be represented directly as a collection of specialized computational units connected in a regular, space-efficient pattern. Data would move through the system at regular "heartbeats" as determined by local state. Figure 1-27 illustrates a design for computing convolutions using a simple linear array. At each beat the input data advances to the right, is multiplied by a local weight, and is accumulated into the output sequence as it also advances to the right. The systolic approach has aspects in common with message passing, data parallel, and dataflow models, but takes on a unique character for a specialized class of problems.

Practical realizations of these ideas, such as iWarp[Bor*90], provided quite general programmability in the nodes, in order for a variety of algorithms to be realized on the same hardware. The key differentiation is that the network can be configured as a collection of dedicated channels, representing the systolic communication pattern, and data can be transferred directly from processor registers to processor registers across a channel. The global knowledge of the communication pattern is exploited to reduce contention and even to avoid deadlock. The key characteristic of systolic architectures is the ability to integrate highly specialized computation under a simple, regular, and highly localized communication patterns.

$$y(i) = w1*x(i) + w2*x(i+1) + w3*x(i+2) + w4*x(i+3)$$
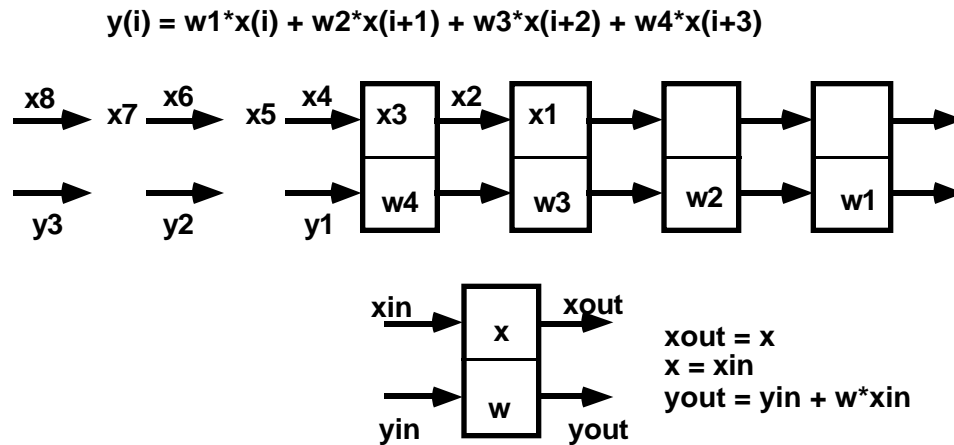
Figure 1-27 Systolic Array computation of an inner product

Systolic algorithms have also been generally amenable to solutions on generic machines, using the fast barrier to delineate coarser grained phases. The regular, local communication pattern of these algorithms yield good locality when large portions of the logical systolic array are executed on each process, the communication bandwidth needed is low, and the synchronization requirements are simple. Thus, these algorithms have proved effective on the entire spectrum of parallel machines.

1.3.7 A Generic Parallel Architecture

In examining the evolution of the major approaches to parallel architecture, we see a clear convergence for scalable machines toward a generic parallel machine organization, illustrated in Figure 1-28. The machine comprises a collection of essentially complete computers, each with one or more processors and memory, connected through a scalable communication network via *communications assist,* some kind of controller or auxiliary processing unit which assists in generating outgoing messages or handling incoming messages. While the consolidation within the field may seem to narrow the interesting design space, in fact, there is great diversity and debate as to what functionality should be provided within the assist and how it interfaces to the processor, memory system, and network. Recognizing that these are specific differences within a largely similar organization helps to understand and evaluate the important organizational trade-offs.

Not surprisingly, different programming models place different requirements on the design of the communication assist, and influence which operations are common and should be optimized. In the shared memory case, the assist is tightly integrated with the memory system in order to capture the memory events that may require interaction with other nodes. Also, the assist must accept messages and perform memory operations and state transitions on behalf of other nodes. In the message passing case, communication is initiated by explicit actions, either at the system or user level, so it is not required that memory system events be observed. Instead, there is a need to initiate the messages quickly and to respond to incoming messages. The response may require that a tag match be performed, that buffers be allocated, that data transfer commence, or that an event be posted. The data parallel and systolic approaches place an emphasis on fast global synchronization, which may be supported directly in the network or in the assist. Dataflow places an emphasis on fast dynamic scheduling of computation based on an incoming message. Systolic algorithms present the opportunity to exploit global patterns in local scheduling. Even with these differences, is important to observe that all of these approaches share common aspects; they need to initiate network transactions as a result of specific processor events, and they need to perform simple operations on the remote node to carry out the desired event.

We also see that a separation has emerged between programming model and machine organization as parallel programming environments have matured. For example, Fortran 90 and High Performance Fortran provide a shared-address data-parallel programming model, which is implemented on a wide range of machines, some supporting a shared physical address space, others with only message passing. The compilation techniques for these machines differ radically, even though the machines appear organizationally similar, because of differences in communication and synchronization operations provided at the user level (i.e., the communication abstraction) and vast differences in the performance characteristics of these communication operations. As a second example, popular message passing libraries, such as PVM (parallel virtual machine) and MPI (message passing interface), are implemented on this same range of machines, but the

implementation of the libraries differ dramatically from one kind of machine to another. The same observations hold for parallel operating systems.
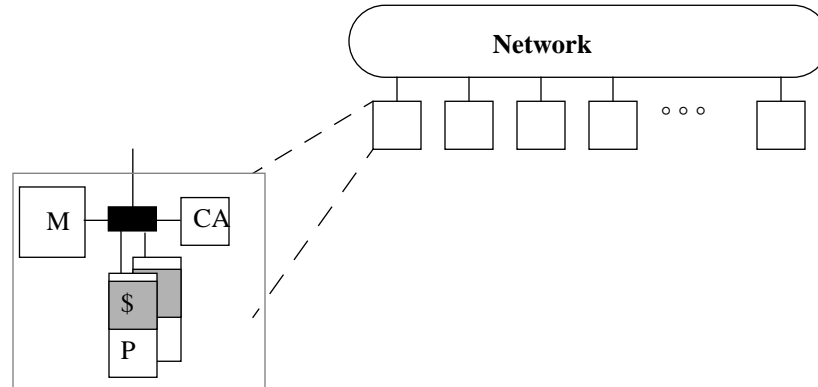
Figure 1-28 Generic scalable multiprocessor organization.

A collection of essentially complete computers, including one or more processors and memory, communicating through a general purpose, high-performance, scalable interconnect. Typically, each node contains a controller which assists in communication operations across the network.

1.4 Fundamental Design Issues

Given how the state of the art in parallel architecture has advanced, we need to take a fresh look at how to organize the body of material in the field. Traditional machine taxonomies, such as SIMD/MIMD, are of little help since multiple general purpose processors are so dominant. One cannot focus entirely on programming models, since in many cases widely differing machine organizations support a common programming model. One cannot just look at hardware structures, since common elements are employed in many different way. instead, we ought to focus our attention on the architectural distinctions that make a difference to the software that is to run on the machine. In particular, we need to highlight those aspects that influence how a compiler should generate code from a high-level parallel language, how a library writer would code a well-optimized library, or how an application would be written in a low-level parallel language. We can then approach the design problem as one that is constrained from above by how programs use the machine and from below by what the basic technology can provide.

In our view, the guiding principles for understanding modern parallel architecture are indicated by the layers of abstraction presented in Figure 1-13. Fundamentally, we must understand the operations that are provided at the user-level communication abstraction, how various programming models are mapped to these primitives, and how these primitives are mapped to the actual hardware. Excessive emphasis on the high-level programming model without attention to how it can be mapped to the machine would detract from understanding the fundamental architectural issues, as would excessive emphasis on the specific hardware mechanisms in each particular machine.

This section looks more closely at the communication abstraction and the basic requirements of a programming model. It then defines more formally the key concepts that tie together the layers:

naming, ordering, and communication and replication of data. Finally, it introduces the basic performance models required to resolve design trade-offs.

1.4.1 Communication Abstraction

The communication abstraction forms the key interface between the programming model and the system implementation. It plays a role very much like the instruction set in conventional sequential computer architecture. Viewed from the software side, it must have a precise, well-defined meaning so that the same program will run correctly on many implementations. Also the operations provided at this layer must be simple, composable entities with clear costs, so that the software can be optimized for performance. Viewed from the hardware side, it also must have a well-defined meaning so that the machine designer can determine where performance optimizations can be performed without violating the software assumptions. While the abstraction needs to be precise, the machine designer would like it not to be overly specific, so it does not prohibit useful techniques for performance enhancement or frustrate efforts to exploit properties of newer technologies.

The communication abstraction is, in effect, a contract between the hardware and the software allowing each the flexibility to improve what it does, while working correctly together. To understand the "terms" of this contract, we need to look more carefully at the basic requirements of a programming model.

1.4.2 Programming Model Requirements

A parallel program consists of one or more threads of control operating on data. A *parallel programming model* specifies what data can be *named* by the threads, what *operations* can be performed on the named data, and what *ordering* exists among these operations.

To make these issues concrete, consider the programming model for a uniprocessor. A thread can name the locations in its virtual address space and can name machine registers. In some systems the address space is broken up into distinct code, stack, and heap segments, while in others it is flat. Similarly, different programming languages provide access to the address space in different ways; for example, some allow pointers and dynamic storage allocation, while others do not. Regardless of these variations, the instruction set provides the operations that can be performed on the named locations. For example, in RISC machines the thread can load data from or store data to memory, but perform arithmetic and comparisons only on data in registers. Older instruction sets support arithmetic on either. Compilers typically mask these differences at the hardware/software boundary, so the user's programing model is one of performing operations on variables which hold data. The hardware translates each virtual address to a physical address on every operation.

The ordering among these operations is *sequential program order* – the programmer's view is that variables are read and modified in the top-to-bottom, left-to-right order specified in the program. More precisely, the value returned by a read to an address is the last value written to the address in the sequential execution order of the program. This ordering assumption is essential to the logic of the program. However, the reads and writes may not actually be performed in program order, because the compiler performs optimizations when translating the program to the instruction set and the hardware performs optimizations when executing the instructions. Both make sure the program cannot tell that the order has been changed. The compiler and hardware

preserve the *dependence order*. If a variable is written and then read later in the program order, they make sure that the later operation uses the proper value, but they may avoid actually writing and reading the value to and from memory or may defer the write until later. Collections of reads with no intervening writes may be completely reordered and, generally, writes to different addresses can be reordered as long as dependences from intervening reads are preserved. This reordering occurs at the compilation level, for example, when the compiler allocates variables to registers, manipulates expressions to improve pipelining, or transforms loops to reduce overhead and improve the data access pattern. It occurs at the machine level when instruction execution is pipelined, multiple instructions are issued per cycle, or when write buffers are used to hide memory latency. We depend on these optimizations for performance. They work because for the program to observe the effect of a write, it must read the variable, and this creates a dependence, which is preserved. Thus, the illusion of program order is preserved while actually executing the program in the looser dependence order.[1] We operate in a world where essentially all programming languages embody a programming model of sequential order of operations on variables in a virtual address space and the system enforces a weaker order wherever it can do so without getting caught.

Now let's return to parallel programming models. The informal discussion earlier in this chapter indicated the distinct positions adopted on naming, operation set, and ordering. Naming and operation set are what typically characterize the models, however, ordering is of key importance. A parallel program must coordinate the activity of its threads to ensure that the dependences within the program are enforced; *this requires explicit synchronization operations when the ordering implicit in the basic operations is not sufficient.* As architects (and compiler writers) we need to understand the ordering properties to see what optimization "tricks" we can play for performance. We can focus on shared address and message passing programming models, since they are the most widely used and other models, such as data parallel, are usually implemented in terms of one of them.

The shared address space programming model assumes one or more threads of control, each operating in an address space which contains a region that is shared between threads, and may contain a region that is private to each thread. Typically, the shared region is shared by all threads. All the operations defined on private addresses are defined on shared addresses, in particular the program accesses and updates shared variables simply by using them in expressions and assignment statements.

Message passing models assume a collection of processes each operating in a private address space and each able to name the other processes. The normal uniprocessor operations are provided on the private address space, in program order. The additional operations, send and receive, operate on the local address space and the global process space. Send transfers data from the local address space to a process. Receive accepts data into the local address space from a process. Each send/receive pair is a specific point-to-point synchronization operation. Many message passing languages offer global or *collective* communication operations as well, such as broadcast.

---

1. The illusion breaks down a little bit for system programmers, say, if the variable is actually a control register on a device. Then the actual program order must be preserved. This is usually accomplished by flagging the variable as special, for example using the volatile type modifier in C.

1.4.3 Naming

The position adopted on naming in the programming model is presented to the programmer through the programming language or programming environment. It is what the logic of the program is based upon. However, the issue of naming is critical at each level of the communication architecture. Certainly one possible strategy is to have the operations in the programming model be one-to-one with the communication abstraction at the user/system boundary and to have this be one-to-one with the hardware primitives. However, it is also possible for the compiler and libraries to provide a level of translation between the programming model and the communication abstraction, or for the operating system to intervene to handle some of the operations at the user/system boundary. These alternatives allow the architect to consider implementing the common, simple operations directly in hardware and supporting the more complex operations partly or wholely in software.

Let us consider the ramification of naming at the layers under the two primary progrramming models: shared address and mssage passing. First, in a shared address model, accesses to shared variables in the program are usually mapped by the compiler to load and store instructions on *shared virtual addresses*, just like access to any other variable. This is not the only option, the compiler could generate special code sequences for accesses to shared variables, the uniform access to private and shared addresses is appealing in many respects. A machine supports a *global physical address space* if any processor is able to generate a physical address for any location in the machine and access the location in a single memory operation. It is straightforward to realize a shared virtual address space on a machine providing a global physical address space: establish the virtual-to-physical mapping so that shared virtual addresses map to the same physical location, i.e., the processes have the same entries in their page tables. However, the existence of the level of translation allows for other approaches. A machine supports *independent local physical address spaces*, if each processor can only access a distinct set of locations. Even on such a machine, a shared virtual address space can be provided by mapping virtual addresses which are local to a process to the corresponding physical address. The non-local addresses are left unmapped, so upon access to a non-local shared address a page fault will occur, allowing the operating system to intervene and access the remote shared data. While this approach can provide the same naming, operations, and ordering to the program, it clearly has different hardware requirements at the hardware/software boundary. The architect's job is resolve these design trade-offs across layers of the system implementation so that the result is efficient and cost effective for the target application workload on available technology.

Secondly, message passing operations could be realized directly in hardware, but the matching and buffering aspects of the send/receive operations are better suited to software implementation. More basic data transport primitives are well supported in hardware. Thus, in essentially all parallel machines, the message passing programming model is realized via a software layer that is built upon a simpler communication abstraction. At the user/system boundary, one approach is to have all message operations go through the operating system, as if they were I/O operations. However, the frequency of message operations is much greater than I/O operations, so it makes sense to use the operating system support to set up resources, priviledges etc. and allow the frequent, simple data transfer operations to be supported directly in hardware. On the other hand, we might consider adopting a shared virtual address space as the lower level communication abstraction, in which case send and receive operations involve writing and reading shared buffers and posting the appropriate synchronization events. The issue of naming arises at each level of abstraction in a parallel architecture, not just in the programming model. As architects, we need to design against the frequency and type of operations that occur at the communication abstrac-

tion, understanding that there are trade-offs at this boundary involving what is supported directly in hardware and what in software.

Operations

Each programming model defines a specific set of operations that can be performed on the data or objects that can be named within the model. For the case of a shared address model, these include reading and writing shared variables, as well as various atomic read-modify-write operations on shared variables, which are used to synchronize the threads. For message passing the operations are send and receive on private (local) addresses and process identifiers, as described above. One can observe that there is a global address space defined by a message passing model. Each element of data in the program is named by a process number and local address within the process. However, there are no operations defined on these global addresses. They can be passed around and interpreted by the program, for example, to emulate a shared address style of programming on top of message passing, but they cannot be operated on directly at the communication abstraction. As architects we need to be aware of the operations defined at each level of abstraction. In particular, we need to be very clear on what ordering among operations is assumed to be present at eacg level of abstraction, where communication takes place, and how data is replicated.

1.4.4 Ordering

The properties of the specified order among operations also has a profound effect throughout the layers of parallel architecture. Notice, for example, that the message passing model places no assumption on the ordering of operations by distinct processes, except the explicit program order associated with the send/receive operations, whereas a shared address model must specify aspects of how processes see the order of operations performed by other processes. Ordering issues are important and rather subtle. Many of the "tricks" that we play for performance in the uniprocessor context involve relaxing the order assumed by the programmer to gain performance, either through parallelism or improved locality or both. Exploiting parallelism and locality is even more important in the multiprocessor case. Thus, we will need to understand what new tricks can be played. We also need to examine what of the old tricks are still valid. Can we perform the traditional sequential optimizations, at the compiler and architecture level, on each process of a parallel program? Where can the explicit synchronization operations be used to allow ordering to be relaxed on the conventional operations? To answer these questions we will need to develop a much more complete understanding of how programs use the communication abstraction, what properties they rely upon, and what machine structures we would like to exploit for performance.

A natural position to adopt on ordering is that operations in a thread are in program order. That is what the programmer would assume for the special case of one thread. However, there remains the question of what ordering can be assumed among operations performed on shared variables by different threads. The threads operate independently and, potentially, at different speeds so there is no clear notion of "latest". If one has in mind that the machines behave as a collection of simple processors operating on a common, centralized memory, then it is reasonable to expect that the global order of memory accesses will be some arbitrary interleaving of the individual program orders. In reality we won't build the machines this way, but it establishes what operations are implicitly ordered by the basic operations in the model. This interleaving is also what we expect if a collection of threads that are timeshared, perhaps at a very fine level, on a uniprocessor.

Where the implicit ordering is not enough, explicit synchronization operations are required. There are two types of synchronization required in parallel programs:

- *Mutual exclusion* ensures that certain operations on certain data are performed by only one thread or process at a time. We can imagine a room that must be entered to perform such an operation, and only one process can be in the room at a time. This is accomplished by locking the door upon entry and unlocking it on exit. If several processes arrive at the door together, only one will get in and the others will wait till it leaves. The order in which the processes are allowed to enter does not matter and may vary from one execution of the program to the next; what matters is that they do so one at a time. Mutual exclusion operations tend to *serialize* the execution of processes.

- *Events* are used to inform other processes that some point of execution has been reached so that they can proceed knowing that certain dependences have been satisfied. These operations are like passing a batton from one runner to the next in a relay race or the starter firing a gun to indicate the start of a race. If one process writes a value which another is supposed to read, there needs to be an event synchronization operation to indicate that the value is ready to be read. Events may be *point-to-point*, involving a pair of processes, or they may be *global*, involving all processes, or a *group* of processes.

1.4.5 Communication and Replication

The final issues that are closely tied to the layers of parallel architecture are that of communication and data replication. Replication and communication are inherently related. Consider first a message passing operation. The effect of the send/receive pair is to copy data that is in the sender's address space into a region of the receiver's address space. This transfer is essential for the receiver to access the data. If the data was produced by the sender, it reflects a *true communication* of information from one process to the other. If the data just happened to be stored at the sender, perhaps because that was the initial configuration of the data or because the data set is simply to large to fit on any one node, then this transfer merely replicates the data to where it is used. The processes are not actually communicating via the data transfer. If the data was replicated or positioned properly over the processes to begin with, there would be no need to communicate it in a message. More importantly, if the receiver uses the data over and over again, it can reuse its replica without additional data transfers. The sender can modify the region of addresses that was previously communicated with no effect on the previous receiver. If the effect of these later updates are to be communicated, an additional transfer must occur.

Consider now a conventional data access on a uniprocessor through a cache. If the cache does not contain the desired address, a miss occurs and the block is transfered from the memory that serves as a backing store. The data is implicitly replicated into cache near the processor that accesses it. If the processor resues the data while it resides in the cache, further transfers with the memory are avoided. In the uniprocessor case, the processor produces the data and the processor that consumes it, so the "communication" with the memory occurs only because the data does not fit in the cache or it is being accessed for the first time.

Interprocess communication and data transfer within the storage hierarchy become melded together in a shared physical address space. Cache misses cause a data transfer across the machine interconnect whenever the physical backing storage for an address is remote to the node accessing the address, whether the address is private or shared and whether the transfer is a result of true communication or just a data access. The natural tendency of the machine is to replicate

data into the caches of the processors that access the data. If the data is reused while it is in the cache, no data transfers occur; this is a major advantage. However, when a write to shared data occurs, something must be done to ensure that later reads by other processors get the new data, rather than the old data that was replicated into their caches. This will involve more than a simple data transfer.

To be clear on the relationship of communication and replication it is important to distinguish several concepts that are frequently bundled together. When a program performs a write, it *binds* a data value to an address; a read obtains the data value bound to an address. The data resides in some physical storage element in the machine. A *data transfer* occurs whenever data in one storage element is transfered into another. This does not necessarily change the bindings of addresses and values. The same data may reside in multiple physical locations, as it does in the uniprocessor storage hierarchy, but the one nearest to the processor is the only one that the processor can observe. If it is updated, the other hidden replicas, including the actual memory location, must eventually be updated. Copying data binds a new set of addresses to the same set of values. Generally, this will cause data transfers. Once the copy has been made, the two sets of bindings are completely independent, unlike the implicit replication that occurs within the storage hierarchy, so updates to one set of addresses do not effect the other. *Communication* between processes occurs when data written by one process is read by another. This may cause a data transfer within the machine, either on the write or the read, or the data transfer may occur for other reasons. Communication may involve establishing a new binding, or not, depending on the particular communication abstraction.

In general, replication avoids "unnecessary" communication, that is transferring data to a consumer that was not produced since the data was previously accessed. The ability to perform replication automatically at a given level of the communication archtiecture depends very strongly on the naming and ordering properties of the layer. Moreover, replication is not a panacea. Replication itself requires data transfers. It is disadvantageous to replicate data that is not going to be used. We will see that replication plays an important role throughout parallel computer architecture.

1.4.6 Performance

In defining the set of operations for communication and cooperation, the data types, and the addressing modes, the communication abstraction specifies how shared objects are named, what ordering properties are preserved, and how synchronization is performed. However, the performance characteristics of the available primitives determines how they are actually used. Programmers and compiler writers will avoid costly operations where possible. In evaluating architectural trade-offs, the decision between feasible alternatives ultimately rests upon the performance they deliver. Thus, to complete our introduction to the fundamental issues of parallel computer architecture, we need to lay a framework for understanding performance at many levels of design.

Fundamentally, there are three performance metrics, *latency*, the time taken for an operation, *bandwidth*, the rate at which operations are performed, and *cost*, the impact these operations have on the execution time of the program. In a simple world where processors do only one thing at a time these metrics are directly related; the bandwidth (operations per second) is the reciprocal of the latency (seconds per operation) and the cost is simply the latency times the number of operations performed. However, modern computer systems do many different operations at once and

the relationship between these performance metrics is much more complex. Consider the following basic example.

Example 1-2   Suppose a component can perform a specific operation in 100ns. Clearly it can support a bandwidth of 10 million operations per second. However, if the component is pipelined internally as ten equal stages, it is able to provide a peak bandwidth of 100 million operations per second. The rate at which operations can be initiated is determined by how long the slowest stage is occupied, 10 ns, rather than by the latency of an individual operation. The bandwidth delivered on an application depends on how frequently it initiates the operations. If the application starts an operation every 200 ns, the delivered bandwidth is 5 million operations per seconds, regardless of how the component is pipelined. Of course, usage of resources is usually birsty, so pipelining can be advantageous even when the average initiation rate is low. If the application performed one hundred million operations on this component, what is the cost of these operations? Taking the operation count times the operation latency would give upper bound of 10 seconds. Taking the operation count divided by the peak rate gives a lower bound of 1 second. The former is accurate if the program waited for each operation to complete before continuing. The latter assumes that the operations are completely overlapped with other useful work, so the cost is simply the cost to initiate the operation. Suppose that on average the program can do 50 ns of useful work after each operation issued to the component before it depends on the operations result. Then the cost to the application is 50 ns per operation – the 10 ns to issue the operation and the 40 ns spent waiting for it to complete.

Since the unique property of parallel computer architecture is communication, the operations that we are concerned with most often are data transfers. The performance of these operations can be understood as a generalization of our basic pipeline example.

Data Transfer Time

The time for a data transfer operation is generally described by a linear model:

$$TransferTime(n) \; = \; T_0 + \frac{n}{B} \qquad\qquad \textbf{(EQ 1.3)}$$

where n is the amount of data (e.g., number of bytes), B is the transfer rate of the component moving the data (e.g., bytes per second), and the constant term, T_0, is the start-up cost. This is a very convenient model, and it is used to describe a diverse collection of operations, including messages, memory accesses, bus transactions, and vector operations. For message passing, the start up cost can be thought of as the time for the first bit to get to the destination. It applies in many aspects of traditional computer architecture, as well. For memory operations, it is essentially the access time. For bus transactions, it reflects the bus arbitration and command phases. For any sort of pipelined operation, including pipelined instruction processing or vector operations, it is the time to fill pipeline.

Using this simple model, it is clear that the bandwidth of a data transfer operation depends on the transfer size. As the transfer size increases it approaches the asymptotic rate of B, which is sometimes referred to as r_∞. How quickly it approaches this rate depends on the start-up cost. It is eas-

ily shown that the size at which half of the peak bandwidth is obtained, the half-power point, is given by

$$n_{\frac{1}{2}} = \frac{T_0}{B}.$$ **(EQ 1.4)**

Unfortunately, this linear model does not give any indication when the next such operation can be initiated, nor does it indicate whether other useful work can be performed during the transfer. These other factors depend on how the transfer is performed.

Overhead and Occupancy

The data transfer in which we are most interested is the one that occurs across the network in parallel machines. It is initiated by the processor through the communication assist. The essential components of this operation can be described by the following simple model.

$$CommunicationTime(n) = Overhead + Network\ Delay + Occupancy$$ **(EQ 1.5)**

The *Overhead* is the time the processor spends initiating the transfer. This may be a fixed cost, if the processor simply has to tell the communication assist to start, or it may be linear in n, if the processor has to copy the data into the assist. The key point is that this is time the processor is busy with the communication event; it cannot do other useful work or initiate other communication during this time. The remaining portions of the communication time is considered the *network latency*; it is the part that can be hidden by other processor operations.

The *Occupancy* is the time it takes for the data to pass through the slowest component on the communication path. For example, each link that is traversed in the network will be occupied for time $\frac{n}{B}$, where B is the bandwidth of the link. The data will occupy other resources, including buffers, switches, and the communication assist. Often the communication assist is the bottleneck that determines the occupancy. The occupancy limits how frequently communication operations can be initiated. The next data transfer will have to wait until the critical resource is no longer occupied before it can use that same resource. If there is buffering between the processor and the bottleneck, the processor may be able to issue a burst of transfers at a frequency greater than $\frac{1}{Occupancy}$, however, once this buffering is full, the processor must slow to the rate set by the occupancy. A new transfer can start only when a older one finishes.

The remaining communication time is lumped into the *Network Delay*, which includes the time for a bit to be routed across the actual network and many other factors, such as the time to get through the communication assist. From the processors viewpoint, the specific hardware components contributing to network delay are indistinguishable. What effects the processor is how long it must wait before it can use the result of a communication event, how much of this time it can be bust with other activities, and how frequently it can communicate data. Of course, the task of designing the network and its interfaces is very concerned with the specific components and their contribution to the aspects of performance that the processor observes.

In the simple case where the processor issues a request and waits for the response, the breakdown of the communication time into its three components is immaterial. All that matters is the total

round trip time. However, in the case where multiple operations are issued in a pipelined fashion, each of the components has a specific influence on the delivered performance.

Indeed, every individual component along the communication path can be described by its delay and its occupancy. The network delay is simply the sum of the delays along the path. The network occupancy is the maximum of the occupancies along the path. For interconnection networks there is an additional factor that arises because many transfers can take place simultaneously. If two of these transfers attempt to use the same resource at once, for example is they use the same wire at the same time, one must wait. This *contention* for resources increases the average communication time. From the processors viewpoint, contention appears as increased occupancy. Some resource in the system is occupied for a time determined by the collection of transfers across it.

Equation 1.5 is a very general model. It can be used to describe data transfers in many places in modern, highly pipelined computer systems. As one example, consider the time to move a block between cache and memory on a miss. There is a period of time that the cache controller spends inspecting the tag to determine that it is not a hit and then starting the transfer; this is the overhead. The occupancy is the block size divided by the bus bandwidth, unless there is some slower component in the system. The delay includes the normal time to arbitrate and gain access to the bus plus the time spent delivering data into the memory. Additional time spent waiting to gain access to the bus or wait for the memory bank cycle to complete is due to contention. A second obvious example is the time to transfer a message from one processor to another.

Communication Cost

The bottom line is, of course, the time a program spends performing communication. A useful model connecting the program characteristics to the hardware performance is given by the following.

$$CommunicationCost \ = \ frequency \times (CommunicationTime - Overlap) \qquad \textbf{(EQ 1.6)}$$

The *frequency of communication*, defined as the number of communication operations per unit of work in the program, depends on many programming factors (as we will see in Chapter 2) and many hardware design factors. In particular, hardware may limit the transfer size and thereby determine the minimum number of messages. It may automatically replicate data or migrate it to where it is used. However, there is a certain amount of communication that is inherent to parallel execution, since data must be shared and processors must coordinate their work. In general, for a machine to support programs with a high communication frequency the other parts of the communication cost equation must be small – low overhead, low network delay, and small occupancy. The attention paid to communication costs essentially determines which programming models a machine can realize efficiently and what portion of the application space it can support. Any parallel computer with good computational performance can support programs that communicate infrequently, but as the frequency increases or volume of communication increases greater stress is placed on the communication architecture.

The *overlap* is the portion of the communication operation which is performed concurrently with other useful work, including computation or other communication. This reduction of the effective cost is possible because much of the communication time involves work done by components of the system other than the processor, such as the network interface unit, the bus, the network or the remote processor or memory. Overlapping communication with other work is a form of small

scale parallelism, as is the instruction level parallelism exploited by fast microprocessors. In effect, we may invest some of the available parallelism in a program to hide the actual cost of communication.

Summary

The issues of naming, operation set, and ordering apply at each level of abtraction in a parallel architecture, not just the programming model. In general, there may be a level of translation or run-time software between the programming model and the communication abstraction, and beneath this abstraction are key hardware abstractions. At any level, communication and replication are deeply related. Whenever two processes access the same data, it either needs to be communicated between the two or replicated so each can access a copy of the data. The ability to have the same name refer to two distinct physical locations in a meaningful manner at a given level of abstraction depends on the position adopted on naming and ordering at that level. Wherever data movement is involved, we need to understand its performance characteristics in terms of the latency and bandwidth, and furthermore how these are influenced by overhead and occupancy. As architects, we need to design against the frequency and type of operations that occur at the communication abstraction, understanding that there are trade-offs across this boundary involving what is supported directly in hardware and what in software. The position adopted on naming, operation set, and ordering at each of these levels has a qualitiative impact on these trade-offs, as we will see throughout the book.

1.5 Concluding Remarks

Parallel computer architecture forms an important thread in the evolution of computer architecture, rooted essentially in the beginnings of computing. For much of this history it takes on a novel, even exotic role, as the avenue for advancement over and beyond what the base technology can provide. Parallel computer designs have demonstrated a rich diversity of structure, usually motivated by specific higher level parallel programming models. However, the dominant technological forces of the VLSI generation have pushed parallelism increasingly into the mainstream, making parallel architecture almost ubiquitous. All modern microprocessors are highly parallel internally, executing several bit-parallel instructions in every cycle and even reordering instructions within the limits of inherent dependences to mitigate the costs of communication with hardware components external to the processor itself. These microprocessors have become the performance and price-performance leaders of the computer industry. From the most powerful supercomputers to departmental servers to the desktop, we see even higher performance systems constructed by utilizing multiple of such processors integrated into a communications fabric. This technological focus, augmented with increasing maturity of compiler technology, has brought about a dramatic convergence in the structural organization of modern parallel machines. The key architectural issue is how communication is integrated into the memory and I/O systems that form the remainder of the computational node. This communications architecture reveals itself functionally in terms of what can be named at the hardware level, what ordering guarantees are provided and how synchronization operations are performed, while from a performance point of view we must understand the inherent latency and bandwidth of the available communication operations. Thus, modern parallel computer architecture carries with it a strong engineering component, amenable to quantitative analysis of cost and performance trade-offs.

Computer systems, whether parallel or sequential, are designed against the requirements and characteristics of intended workloads. For conventional computers, we assume that most practitioners in the field have a good understanding of what sequential programs look like, how they

are compiled, and what level of optimization is reasonable to assume the programmer has performed. Thus, we are comfortable taking popular sequential programs, compiling them for a target architecture, and drawing conclusions from running the programs or evaluating execution traces. When we attempt to improve performance through architectural enhancements, we assume that the program is reasonably "good" in the first place.

The situation with parallel computers is quite different. There is much less general understanding of the process of parallel programming and there is a wider scope for programmer and compiler optimizations, which can greatly affect the program characteristics exhibited at the machine level. To address this situation, Chapter 2 provides an overview of parallel programs, what they look like, how they are constructed. Chapter 3 explains the issues that must be addressed by the programmer and compiler to construct a "good" parallel program, i.e., one that is effective enough is using multiple processors to form a reasonable basis for architectural evaluation. Ultimately, we design parallel computers against the program characteristics at the machine level, so the goal of Chapter 3 is to draw a connection between what appears in the program text and how the machine spends its time. In effect, Chapter 2 and 3 take us from a general understanding of issues at the application level down to a specific understanding of the character and frequency of operations at the communication abstraction.

Chapter 4 establishes a framework for workload-driven evaluation of parallel computer designs. Two related scenarios are addressed. First, for a parallel machine that has already been built, we need a sound method of evaluating its performance. This proceeds by first testing what individual aspects of the machine are capable of in isolation and then measures how well they perform collectively. The understanding of application characteristics is important to ensure that the workload run on the machine stresses the various aspects of interest. Second, we outline a process of evaluating hypothetical architectural advancements. New ideas for which no machine exists need to be evaluated through simulations, which imposes severe restrictions on what can reasonably be executed. Again, an understanding of application characteristics and how they scale with problem and machine size is crucial to navigating the design space.

Chapters 5 and 6 study the design of symmetric multiprocessors with a shared physical address space in detail. There are several reasons to go deeply into the small-scale case before examining scalable designs. First, small-scale multiprocessors are the most prevalent form of parallel architecture; they are likely to be what the most students are exposed to, what the most software developers are targeting and what the most professional designers are dealing with. Second, the issues that arise in the small-scale are indicative of what is critical in the large scale, but the solutions are often simpler and easier to grasp. Thus, these chapters provides a study in the small of what the following four chapters address in the large. Third, the small-scale multiprocessor design is a fundamental building block for the larger scale machines. The available options for interfacing a scalable interconnect with a processor-memory node are largely circumscribed by the processor, cache, and memory structure of the small scale machines. Finally, the solutions to key design problems in the small-scale case are elegant in their own right.

The fundamental building-block for the designs in Chapter 5 and 6 is the shared bus between processors and memory. The basic problem that we need to solve is to keep the contents of the caches coherent and the view of memory provided to the processors consistent. A bus is a powerful mechanism. It provides any-to-any communication through a single set of wires, but moreover can serve as a broadcast medium, since there is only one set of wires, and even provide global status, via wired-or signals. The properties of bus transactions are exploited in designing extensions of conventional cache controllers that solve the coherence problem. Chapter 5 pre-

sents the fundamental techniques to bus-based cache coherence at the logical level and presents the basic design alternatives. These design alternatives provide an illustration of how workload-driven evaluation of can be brought to bear in making design decisions. Finally, we return to the parallel programming issues of the earlier chpapters and examine how aspects of the machine design influence the software level, especially in regard to cache effects on sharing patterns and the design of robust synchronization routines. Chapter 6 focuses on the organization structure and machine implementation of bus-based cache coherence. It examines a variety of more advanced designs that seek to reduce latency and increase bandwidth while preserving a consistent view of memory.

Chapters 7 through 10 form a closely interlocking study of the design of scalable parallel architectures. Chapter 7 makes the conceptual step from a bus transaction as a building block for higher level abstractions to a *network transaction* as a building block. To cement this understanding, the communication abstractions that we have surveyed in this introductory chapter are constructed from primitive network transactions. Then the chapter studies the design of the node to network interface in depth using a spectrum of case studies.

Chapters 8 amd 9 go deeply into the design of scalable machines supporting a shared address space, both a shared physical address space and a shared virtual address space upon independent physical address spaces. The central issue is automatic replication of data while preserving a consistent view of memory and avoiding performance bottlenecks. The study of a global physical address space emphasizes hardware organizations that provide efficient, fine-grain sharing. The study of a global virtual address space provides an understanding of what is the minimal degree of hardware support required for most workloads.

Chapter 10 takes up the question of the design of the scalable network itself. As with processors, caches, and memory systems, there are several dimensions to the network design space and often a design decision involves interactions along several dimensions. The chapter lays out the fundamental design issues for scalable interconnects, illustrates the common design choices, and evaluates them relative to the requirements established in chapters 5 and 6. Chapter 9 draws together the material from the other three chapters in the context of an examination of techniques for latency tolerance, including bulk transfer, write-behind, and read-ahead across the spectrum of communication abstractions. Finally, Chapter 11 looks at what has been learned throughout the text in light of technologocal, application, and economic trends to forcast what will be the key on-going developments in parallel computer architecture. The book presents the conceptual foundations as well as the engineering issues across a broad range of potential scales of design, all of which have an important role in computing today and in the future.

1.6 References

[AlGo89]   George S. Almasi and Alan Gottlieb. Highly Parallel Computing. Benjamin/Cummings, Redwood CA 1989.

[And*62]   J. P. Anderson, S. A. Hoffman, J. Shifman, and R. Williams, "D825 - a Mulitple-computer System for Command and Control", AFIP Proceedings of the FJCC, vol 22, pp. 86-96, 1962.

[Amdh67]   Gene M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS 1967 Spring Joint Computer Conference, vol 40, pp. 483-5.

[Don94]   Jack J. Dongarra. Performance of Various Computers Using Standard Linear Equation Software. Tech Report CS-89-85, Computer Science Dept., Univ. of Tennessee, Nov. 1994 (current report available from netlib@ornl.gov).

[Fly72]   M. J. Flynn. Some Computer Organizations and Their Effectiveness. IEEE Trans. on Computing, C-21():948-60.

[HeP90]   John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 1990.

[HoJe]   R. W. Hockney and C. R. Jesshope. Parallel Computers 2. Adam Hilger, 1988.

Technology and Application Trends

[Bak90]   H. B. Bakoglu, Circuits, Interconnections, and Packaging for VLSI. Addison-Wesley Pub. Co., 1990.

[Fei94]   Curtis P. Feigel, TI Introduces Four-Processor DSP Chip, Microprocessor Report, Mar 28, 1994.

[Fla94]   James L. Flanagan. Technologies for Multimedia Communications. IEEE Proceedings, pp. 590-603, vol 82, No. 4, April 1994

[Gwe94a]   Linley Gwennap, PA-7200 Enables Inexpensive MP Systems, Microprocessor Report, Mar. 1994. (This should be replaced by Compon94 or ISSCC reference.)

[Gwe94b]   Linley Gwennap, Microprocessors Head Toward MP on a Chip, Microprocessor Report, May 9, 1994.

[HeJo91]   John Hennessy and Norm Jouppi, Computer Technology and Architecture: An Evolving Interaction. IEEE Computer, 24(9):18-29, Sep. 1991.

[MR*]   Several issues of Microprocessor report provide data for chip clock-rate and frequency. Need to flesh this out.

[OST93]   Office of Science and Technology Policy. Grand Challenges 1993: High Performance Computing and Communications, A Report by the Committee on Physical, Mathematical, and Engineering Sciences, 1993.

[Sla94]   Michael Slater, Intel Unveils Multiprocessor System Specification, Microprocessor Report, pp. 12-14, May 1994.

[SPEC]   Standard Performance Evaluation Corporation, http://www.specbench.org/

[Russ78]   R. M. Russel, The CRAY-1 Computer System, Communications of the ACM, 21(1):63-72, Jan 1978

[Aug*89]   August, M.C.; Brost, G.M.; Hsiung, C.C.; Schiffleger, A.J. Cray X-MP: the birth of a supercomputer, Computer, Jan. 1989, vol.22, (no.1):45-52.

[NAS]   Bailey, D.H.; Barszcz, E.; Dagum, L.; Simon, H.D. NAS Parallel Benchmark results 3-94, Proceedings of the Scalable High-Performance Computing Conference, Knoxville, TN, USA, 23-25 May 1994, p. 111-20.

[JNNIE]   Pfeiffer, Hotovy, S., Nystrom, N., Rudy, D., Sterling, T., Straka, "JNNIE: The Joint NSF-NASA Initiative on Evaluation", URL http://www.tc.cornell.edu/JNNIE/finrep/jnnie.html.

Instruction Level Parallelism

[But+91]   M. Butler, T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single Instruction Stream Parallelism is Greater than Two, pp. 276-86, ISCA 1991.

[Cha+91]   P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, W. W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction Issue Processors, pp. 226-ISCA 1991.

[Hor+90]   R. W. Horst, R. L. Harris, R. L. Jardine. Multiple Instruction Issue in the NonStop Cyclone Processor, pp. 216-26, ISCA 1990.

[Joh90]   M. Johnson. Superscalar Processor Design, Prentice Hall, Englewood Cliffs, NJ, 1990.

[JoWa89]   N. Jouppi and D. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, ASPLOS III, pp. 272-282, 1989

[LaWi92]   Monica S. Lam and Robert P. Wilson, Limits on Control Flow on Parallelism, Proc. of the 19th Annual International Symposium on Computer Architecture, pp. 46-57.

[Lee+91]   R. L. Lee, A. Y. Kwok, and F. A. Briggs. The Floating Point Performance of a Superscalar SPARC Processor, ASPLOS IV, Apr. 1991.

[MePa91]   S. Melvin and Y. Patt. Exploiting Fine-Grained Parallelism through a Combination of Hardware and Software Techniques, pp. 287-296, ISCA 1991.

[Smi+89]   M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on Multiple Instruction Issue, Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 290-302, Apr. 1989.

[Soh*94]   G. Sohi, S. Breach, and T. N. Vijaykumar, Multiscalar Processors, Proc. of the 22nd Annual International Symposium on Computer Architecture, Jun 1995, pp. 414-25.

[Wal91]   David W. Wall. Limits of Instruction-Level Parallelism, ASPLOS IV, Apr. 1991.

Shared Memory

[Loki61]   W. Lonergan and P. King, Design of the B 5000 System. Datamation, vol 7 no. 5, may 1961, pp. 28-32

[Pad81]   A. Padegs, System/360 and Beyond, IBM Journal of Research and Development, Vol 25., no. 5, Sept. 1981, pp377-90.

[ArBa86]   James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[Bel85]   C. Gordon Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462-467, April 1985.

[Bro*??]   E. D. Brooks III, B. C. Gorda, K. H. Warren, T. S. Welcome, BBN TC2000 Architecture and Programming Models.

[Goo83]   James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on   Computer Architecture*, pp. 124-131, June 1983.

[Got*83]   A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, IEEE Trans. on Computers, c-32(2):175-89, Feb. 1983.

[HEL+86]   Mark Hill et al. Design Decisions in SPUR. *IEEE Computer*, 19(10):8-22, November 1986.

[KeSc93]   R. E. Kessler and J. L. Schwarzmeier, Cray T3D: a new dimension for Cray Research, Proc. of Papers. COMPCON Spring'93, pp 176-82, San Francisco, CA, Feb. 1993.

[Koe*94]   R. Kent Koeninger, Mark Furtney, and Martin Walker. A Shared Memory MPP from Cray Research, Digital Technical Journal 6(2):8-21, Spring 1994.

[LLJ+92]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. IEEE Transactions on Parallel and Distributed Systems, 4(1):41-61, January 1993.

[Pfi*85]   G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliff, E. A. Melton, V. A. Norton, and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, International Conference on Parallel Processing, 1985.

[Swa*77a]   R. J. Swan, S. H. Fuller, and D. P. Siewiorek, CM* - a modular, multi-microprocessor, AFIPS Conference Proceedings, vol 46, National Computer Conference, 1977, pp. 637-44.

[Swa*77b]   R. J. Swan, A. Bechtolsheim, K-W Lai, and J. K. Ousterhout, The Implementation of the CM* multi-microprocessor, AFIPS Conference Proceedings, vol 46, National Computer Conference, 1977, pp. 645-55.

[Wul*75]   W. Wulf, R. Levin, and C. Person, Overview of the Hydra Operating System Development, Proc. of the 5th Symposium on Operating Systems Principles, Nov. 1975, pp. 122-131.

[AR94]   Thomas B. Alexander, Kenneth G. Robertson, Deal T. Lindsay, Donald L. Rogers, John R. Obermeyer, John R. Keller, Keith Y. Oka, and Marlin M. Jones II. Corporate Business Servers: An Alternative to Mainframes for Business Computing. Hewlett-Packard Journal, June 1994, pages 8-33. (HP K-class)

[Fen*95]   David M. Fenwick, Denis J. Foley, William B. Gist, Stephen R. VanDoren, and Daniel Wissell, The AlphaServer 8000 Series: High-end Server Platform Development, Digital Technical Journal, Vol. 7 No. 1, 1995.

[GoMa95]   Nitin D. Godiwala and Barry A. Maskas, The Second-generation Processor Module for AlphaServer 2100 Systems, Digital Technical Journal, Vol. 7 No. 1, 1995.

[Rod85]   D. Rodgers, Improvements ion Multiprocessor System Design, Proc. of the 12th Annual International Symposium on Computer Architecture, pp. 225-31, Jun1985, Boston MA. (Sequent B8000)

[Nes85]   E. Nestle and A Inselberg, The Synapse N+1 System: Architectural Characteristics and Performance Data of a Tightly-coupled Multiprocessor System, Proc. of the 12th Annual International Symposium on Computer Architecture, pp. 233-9, June 1985, Boston MA. (Synapse)

[Sha85]   Schanin, D.J., The design and development of a very high speed system bus-the Encore Multimax Nanobus, 1986 Proceedings of the Fall Joint Computer Conference. Dallas, TX, USA, 2-6 Nov. 1986). Edited by: Stone, H.S. Washington, DC, USA: IEEE Comput. Soc. Press, 1986. p. 410-18. (Encore)

[Mate85]   N. Matelan, The FLEX/32 Multicomputer, Proc. of the 12th Annual International Symposium on Computer Architecture, pp. 209-13, Jun1985, Boston MA. (Flex)

[Sav95]   J. Savage, Parallel Processing as a Language Design Problem, Proc. of the 12th Annual International Symposium on Computer Architecture, pp. 221-224, Jun 1985, Boston MA. (Myrias 4000)

[HoCh85]   R. W. Horst and T. C. K. Chou, An Architecture for High Volume Transaction Processing, Proc. of the 12th Annual International Symposium on Computer Architecture, pp. 240-5, June 1985, Boston MA. (Tandem NonStop II)

[Cek*93]   Cekleov, M., et al, SPARCcenter 2000: multiprocessing for the 90's, Digest of Papers. COMPCON Spring'93 San Francisco, CA, USA, 22-26 Feb. 1993). Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1993. p. 345-53. (Sun)

[Gal*94]   Galles, M.; Williams, E. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences Vol. I: Architecture, Wailea, HI, USA, 4-7 Jan. 1994, Edited by: Mudge, T.N.; Shriver, B.D. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994. p. 134-43. (SGI Challenge)

[Aga*94]   Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Ben-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. Proc. of the 22nd Annual International Symposium on Computer Architecture, May 1995, pp. 2-13.

[Fra93]   Frank, S.; Burkhardt, H., III; Rothnie, J. The KSR 1: bridging the gap between shared memory and MPPs, COMPCON Spring '93. Digest of Papers, San Francisco, CA, USA, 22-26 Feb. 1993) p. 285-94. (KSR)

[Saa93]   Saavedra, R.H.; Gains, R.S.; Carlton, M.J. Micro benchmark analysis of the KSR1, Proceedings SUPERCOMPUTING '93, Portland, OR, USA, 15-19 Nov. 1993 p. 202-13.

[Len*92]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In Proceedings of the 19th International Symposium on Computer Architecture, pages 92-103, Gold Coast, Australia, May 1992

[IEE93]   IEEE Computer Society. IEEE Standard for Scalable Coherent Interface (SCI). IEEE Standard 1596-1992, New York, August 1993.

[Woo*93]   David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, Steven K. Reinhardt, Mechanisms for Cooperative Shared Memory, Proc. 20th Annual Symposium on Computer Architecture, May 1993, pp. 156-167.

Message Passing

[AtSe88]   William C. Athas and Charles L. Seitz, Multicomputers: Message-Passing Concurrent Computers, IEEE Computer, Aug. 1988, pp. 9-24.

[Bar*94]   E. Barton, J. Crownie, and M. McLaren. Message Passing on the Meiko CS-2. Parallel Computing, 20(4):497-507, Apr. 1994.

[BoRo89]   Luc Bomans and Dirk Roose, Benchmarking the iPSC/2 Hypercube Multiprocessor. Concurrency: Practice and Experience, 1(1):3-18, Sep. 1989.

[Dun88]   T. H. Dunigan, Performance of a Second Generation Hypercube, Technical Report ORNL/TM-10881, Oak Ridge Nat. Lab., Nov. 1988.

[Fox*88]   G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems on Concurrent Processors, vol 1, Prentice-Hall, 1988.

[Gro92]   W. Groscup, The Intel Paragon XP/S supercomputer. Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Nov 1992, pp. 262--273.

[HoMc93]   Mark Homewood and Moray McLaren, Meiko CS-2 Interconnect Elan - Elite Design, Hot Interconnects, Aug. 1993.

[Hor*93]   Horiw, T., Hayashi, K., Shimizu, T., and Ishihata, H., Improving the AP1000 Parallel Computer Performance with Message Passing, Proc. of the 20th Annual International Symposium on Computer Architecture, May 1993, pp. 314-325

[Kum92]   Kumar, M., Unique design concepts in GF11 and their impact on performance, IBM Journal of Research and Development, Nov. 1992, vol.36, (no.6):990-1000.

[Lei*92]   C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, R. Zak. The Network Architecture of the CM-5. Symposium on Parallel and Distributed Algorithms'92, Jun 1992, pp. 272-285.

[Miy94]   Miyoshi, H.; Fukuda, M.; Iwamiya, T.; Takamura, T., et al, Development and achievement of NAL Numerical Wind Tunnel (NWT) for CFD computations. Proceedings of Supercomputing '94, Washington, DC, USA, 14-18 Nov. 1994) p. 685-92.

[PiRe94]   Paul Pierce and Greg Regnier, The Paragon Implementation of the NX Message Passing Interface. Proc. of the Scalable High-Performance Computing Conference, pp. 184-90, May 1994.

[Sei85]   Charles L. Seitz, The Cosmic Cube, Communications of the ACM, 28(1):22-33, Jan 1985.

Networks of Workstations

[And*94]   Anderson, T.E.; Culler, D.E.; Patterson, D. A case for NOW (Networks of Workstations), IEEE Micro, Feb. 1995, vol.15, (no.1):54-6

[Kun*91]   Kung, H.T, et al, Network-based multicomputers: an emerging parallel architecture, Proceedings Supercomputing '91, Albuquerque, NM, USA, 18-22 Nov. 1991 p. 664-73

[Pfi*95]   G. F. Pfister, In Search of Clusters - the coming battle for lowly parallel computing, Prentice-Hall, 1995.

[Bod*95]   N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su, Myrinet: A Gigabit-per-Second Local Area Network, IEEE Micro, Feb. 1995, vol.15, (no.1), pp. 29-38.

[Gil96]   Richard Gillett, Memory Channel Network for PCI, IEEE Micro, vol 16, no. 1, feb 1996.

Data Parallel / SIMD / Systolic

[Bal*62]   J. R. Ball, R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, D. H. Shaffer, On the Use of the Solomon Parallel-Processing Computer, Proceedings of the AFIPS Fall Joint Computer Conference, vol. 22, pp. 137-146, 1962.

[Bat79]   Kenneth E. Batcher, The STARAN Computer. Infotech State of the Art Report: Supercomputers. vol 2, ed C. R. Jesshope and R. W. Hockney, Maidenhead: Infotech Intl. Ltd, pp. 33-49.

[Bat80]   Kenneth E. Batcher, Design of a Massively Parallel Processor, IEEE Transactions on Computers, c-29(9):836-840.

[Bou*72]   W. J. Bouknight, S. A Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, The Illiac IV System, Proceedings of the IEEE, 60(4):369-388, Apr. 1972.

[Bor*90]   Borkar, S. et al, Supporting systolic and memory communication in iWarp, Proceedings. The 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, 28-31 May 1990, p. 70-81.

[Cor72]   J. A. Cornell. Parallel Processing of ballistic missile defense radar data with PEPE, COMPCON 72, pp. 69-72.

[Hill85]   W. Daniel Hillis, The Connection Machine, MIT Press 1985.

[HiSt86]   W. D. Hillis and G. L. Steele, Data-Parallel Algorithms, Comm. ACM, vol. 29, no. 12, 1986

[Nic90]   Nickolls, J.R., The design of the MasPar MP-1: a cost effective massively parallel computer. COMPCON Spring '90 Digest of Papers. San Francisco, CA, USA, 26 Feb.-2 March 1990, p. 25-8.

[Red73]   S. F. Reddaway, DAP - a distributed array processor. First Annual International Symposium on Computer Architecture, 1973.

[Slo*62]   Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds, The Solomon Computer, Proceed-

ings of the AFIPS Fall Joint Computer Conference, vol. 22, pp. 97-107, 1962.

[Slot67]   Daniel L. Slotnick, Unconventional Systems, Proceedings of the AFIPS Spring Joint Computer Conference, vol 30, pp. 477-81, 1967.

[TuRo88]   L.W. Tucker and G.G. Robertson, Architecture and Applications of the Connection Machine, IEEE Computer, Aug 1988, pp. 26-38

[ViCo78]   C.R. Vick and J. A. Cornell, PEPE Architecture - present and future. AFIPS Conference Proceedings, 47: 981-1002, 1978.

Dataflow / Message Driven

[ArCu86]   Dataflow Architectures, Annual Reviews in Computer Science, Vol 1., pp 225-253, Annual Reviews Inc., Palo Alto CA, 1986 (Reprinted in Dataflow and Reduction Architectures, S. S. Thakkar, ed, IEEE Computer Society Press, 1987).

[Dal93]   Dally, W.J.; Keen, J.S.; Noakes, M.D., The J-Machine architecture and evaluation, Digest of Papers. COMPCON Spring '93, San Francisco, CA, USA, 22-26 Feb. 1993, p. 183-8.

[Den80]   J. B. Dennis. Dataflow Supercomputers, IEEE Computer 13(11):93-100.

[GrHo90]   Grafe, V.G.; Hoch, J.E., The Epsilon-2 hybrid dataflow architecture, COMPCON Spring '90, San Francisco, CA, USA, 26 Feb.-2 March 1990, p. 88-93.

[Gur*85]   J. R. Gurd, C. C. Kerkham, and I. Watson, The Manchester Prototype Dataflow Computer, Communications of the ACM, 28(1):34-52, Jan. 1985.

PaCu90]   Papadopoulos, G.M. and Culler, D.E. Monsoon: an explicit token-store architecture, Proceedings. The 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, 28-31 May 1990, p. 82-91.

[Sak*91]   Sakai, S.; Kodama, Y.; Yamaguchi, Y., Prototype implementation of a highly parallel dataflow machine EM4. Proceedings. The Fifth International Parallel Processing Symposium, Anaheim, CA, USA, 30 April-2 May, p. 278-86.

[Shi*84]   T. Shimada, K. Hiraki, and K. Nishida, An Architecture of a Data Flow Machine and its Evaluation, Proc. of Compcon 84, pp. 486-90, IEEE, 1984.

[Spe*93]   E.Spertus, S. Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, William J. Dally, Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5, Proc. 20th Annual Symposium on Computer Architecture, May 1993, 302-313

[vEi*92]   von Eicken, T.; Culler, D.E.; Goldstein, S.C.; Schauser, K.E., Active messages: a mechanism for integrated communication and computation, Proc. of 19th Annual International Symposium on Computer Architecture, old Coast, Queensland, Australia, 19-21 May 1992, pp. 256-66

Historical References

Parallel computer architecture has a long, rich, and varied history that is deeply interwoven with advances in the underlying processor, memory, and network technologies. The first blossoming of parallel architectures occurs around 1960. This is a point where transistors have replaced tubes and other complicated and constraining logic technologies. Processors are smaller and more manageable. A relatively cheap, inexpensive storage technology exists (core memory), and computer architectures are settling down into meaningful "families." Small-scale shared-memory multiprocessors took on an important commercial role at this point with the inception of what we call mainframes today, including the Burroughs B5000[LoKi61] and D825[And*62] and the IBM System 360 model 65 and 67[Pad81]. Indeed, support for multiprocessor configurations was one of the key extensions in the evolution of the 360 architecture to System 370. These included atomic memory opera-

tions and interprocessor interrupts. In the scientific computing area, shared-memory multiprocessors were also common. The CDC 6600 provided an asymmetric shared-memory organization to connect multiple peripheral processors with the central processor, and a dual CPU configuration of this machine was produced. The origins of message-passing machines can be seen as the RW400, introduced in 1960[Por60]. Data parallel machines also emerged, with the design of the Solomon computer[Bal*62,Slo*62].

Up through the late 60s there was tremendous innovation in the use of parallelism within the processor through pipelining and replication of function units to obtain a far greater range of performance within a family than could be obtained by simply increasing the clock rate. It was argued that these efforts were reaching a point of diminishing returns and a major research project got underway involving the University of Illinois and Burroughs to design and build a 64 processor SIMD machine, called Illiac IV[Bou*67], based on the earlier Solomon work (and in spite of Amdahl's arguments to the contrary[Amdh67]). This project was very ambitious, involving research in the basic hardware technologies, architecture, I/O devices, operating systems, programming languages, and applications. By the time a scaled-down, 16 processor system was working in 1975, the computer industry had undergone massive structural change.

First, the concept of storage as a simple linear array of moderately slow physical devices had been revolutionized, first with the idea of virtual memory and then with the concept of caching. Work on Multics and its predecessors, e.g., Atlas and CTSS, separated the concept of the user address space from the physical memory of the machine. This required maintaining a short list of recent translations, a TLB, in order to obtain reasonable performance. Maurice Wilkes, the designer of EDSAC, saw this as a powerful technique for organizing the addressable storage itself, giving rise to what we now call the cache. This proved an interesting example of locality triumphing over parallelism. The introduction of caches into the 360/85 yielded higher performance than the 360/91, which had a faster clock rate, faster memory, and elaborate pipelined instruction execution with dynamic scheduling. The use of caches was commercialized in the IBM 360/185, but this raised a serious difficulty for the I/O controllers as well as the additional processors. If addresses were cached and therefore not bound to a particular memory location, how was an access from another processor or controller to locate the valid data? One solution was to maintain a directory of the location of each cache line. An idea that has regained importance in recent years.

Second, storage technology itself underwent a revolution with semiconductor memories replacing core memories. Initially, this technology was most applicable to small cache memories. Other machines, such as the CDC 7600, simply provided a separate, small, fast explicitly addressed memory. Third, integrated circuits took hold. The combined result was that uniprocessor systems enjoyed a dramatic advance in performance, which mitigated much of the added value of parallelism in the Illiac IV system, with its inferior technological and architectural base. Pipelined vector processing in the CDC STAR-100 addressed the class of numerical computations that Illiac was intended to solve, but eliminated the difficult data movement operations. The final straw was the introduction of the Cray-1 system, with an astounding 80 MHz clock rate owing to exquisite circuit design and the use of what we now call a RISC instruction set, augmented with vector operations using vector registers and offering high peak rate with very low start-up cost. The use of simple vector processing coupled with fast, expensive ECL circuits was to dominate high performance computing for the next 15 years.

A fourth dramatic change occurred in the early 70s, however, with the introduction of microprocessors. Although the performance of the early microprocessors was quite low, the improvements were dramatic as bit-slice designs gave way to 4-bit, 8-bit, 16-bit and full-word designs. The potential of this technology motivated a major research effort at Carnegie-Mellon University to design a large shared memory multiprocessor using the LSI-11 version of the popular PDP-11 minicomputer. This project went through two phases. First, C.mmp connected 16 processors through a specially design circuit-switched cross-bar to a collection of memories and I/O devices, much like the dancehall design in Figure 1-19a[Wul*75]. Second, CM* sought to build a hundred

processor system by connecting 14-node clusters with local memory through a packet-switched network in a NUMA configuration[Swa*77a,Swa77b], as in Figure 1-19b.

This trend toward systems constructed from many, small microprocessors literally exploded in the early to mid 80s. This resulted in the emergence of several disparate factions. On the shared memory side, it was observed that a confluence of caches and properties of busses made modest multiprocessors very attractive. Busses have limited bandwidth, but are a broadcast medium. Caches filter bandwidth and provide a intermediary between the processor and the memory system. Research at Berkeley [Goo83,Hell*86] introduced extensions of the basic bus protocol that allowed the caches to maintain a consistent state. This direction was picked up by several small companies, including Synapse[Nes85], Sequent[Rod85], Encore[Bel85,Sha85], Flex[Mate85] and others, as the 32-bit microprocessor made its debut and the vast personal computer industry took off. A decade later this general approach dominates the server and high-end workstation market and is taking hold in the PC servers and the desktop. The approach experienced a temporary set back as very fast RISC microprocessors took away the performance edge of multiple slower processors. Although the RISC micros were well suited to multiprocessor design, their bandwidth demands severely limited scaling until a new generation of shared bus designs emerged in the early 90s.

Simultaneously, the message passing direction took off with two major research efforts. At CalTech a project was started to construct a 64-processor system using i8086/8087 microprocessors assembled in a hypercube configuration[Sei85,AtSe88]. From this base-line several further designs were pursued at CalTech and JPL[Fox*88] and at least two companies pushed the approach into commercialization, Intel with the iPSC series[] and Ametek. A somewhat more aggressive approach was widely promoted by the INMOS corporation in England in the form of the Transputer, which integrated four communication channels directly onto the microprocessor. This approach was also followed by nCUBE, with a series of very large scale message passing machines. Intel carried the commodity processor approach forward, replacing the i80386 with the faster i860, then replacing the network with a fast grid-based interconnect in the Delta[] and adding dedicated message processors in the Paragon. Meiko moved away from the transputer to the i860 in their computing surface. IBM also investigated an i860-based design in Vulcan.

Data parallel systems also took off in the early 80s, after a period of relative quiet. This included Batcher's MPP system developed by Goodyear for image processing and the Connection Machine promoted by Hillis for AI Applications[Hill85]. The key enhancement was the provision of a general purpose interconnect to problems demanding other than simple grid-based communication. These ideas saw commercialization with the emergence of Thinking Machines Corporation, first with the CM-1 which was close to Hillis' original conceptions and the CM-2 which incorporated a large number of bit-parallel floating-point units. In addition, MASPAR and Wavetracer carried the bit-serial, or slightly wider organization forward in cost-effective systems.

A more formal development of highly regular parallel systems emerged in the early 80s as systolic arrays, generally under the assumption that a large number of very simple processing elements would fit on a single chip. It was envisioned that these would provide cheap, high performance special-purpose add-ons to conventional computer systems. To some extent these ideas have been employed in programming data parallel machines. The iWARP project at CMU produced a more general, smaller scale building block which has been developed further in conjunction with Intel. These ideas have also found their way into fast graphics, compression, and rendering chips.

The technological possibilities of the VLSI revolution also prompted the investigation of more radical architectural concepts, including dataflow architectures[Den80,Arv*83,Gur*85], which integrated the network very closely with the instruction scheduling mechanism of the processor. It was argued that very fast dynamic scheduling throughout the machine would hide the long communication latency and synchronization costs of a

large machine and thereby vastly simplify programming. The evolution of these ideas tended to converge with the evolution of message passing architectures, in the form of message driven computation [Dal93]

Large scale shared-memory designs took off as well. IBM pursued a high profile research effort with the RP-3[Pfi*85] which sought to connect a large number of early RISC processors, the 801, through a butterfly network. This was based on the NYU Ultracomputer work[Gott*83], which was particularly novel for its use of combining operations. BBN developed two large scale designs, the BBN Butterfly using Motorola 68000 processors and the TC2000[Bro*] using the 88100s. These efforts prompted a very broad investigation of the possibility of providing cache-coherent shared memory in a scalable setting. The Dash project at Stanford sought to provide a fully cache coherent distributed shared memory by maintaining a directory containing the disposition of every cache block[LLJ*92,Len*92]. SCI represented an effort to standardize an interconnect and cache-coherency protocol[IEEE93]. The Alewife project at MIT sought to minimize the hardware support for shared memory[Aga*94], which was pushed further by researchers ar Wisconsin[Woo*93]. The Kendall Square Research KSR1[Fra93,Saa93] goes even further and allows the home location of data in memory to migrate. Alternatively, the Denelcor HEP attempted to hide the cost of remote memory latency by interleaving many independent threads on each processor.

The 90s have exhibited the beginnings of a dramatic convergence among these various factions. This convergence is driven by many factors. One is clearly that all of the approach have clear common aspects. They all require a fast, high quality interconnect. They all profit from avoiding latency where possible and reducing the absolute latency when it does occur. They all benefit from hiding as much of the communication cost as possible, where it does occur. They all must support various forms of synchronization. We have seen the shared memory work explicit seek to better integrate message passing in Alewife[Aga*94] and Flash [Flash-ISCA94], to obtain better performance where the regularity of the application can provide large transfers. We have seen data parallel designs incorporate complete commodity processors in the CM-5[Lei*92], allowing very simple processing of messages at user level, which provides much better efficiency for Message Driven computing and shared memory[vEi*92,Spe*93]. There remains the additional support for fast global synchronization. We have seen fast global synchronization, message queues, and latency hiding techniques developed in a NUMA shared memory context in the Cray T3D[Kesc93,Koe*94] and the message passing support in the Meiko CS-2[Bar*94,HoMc93] provides direct virtual memory to virtual memory transfers within the user address space. The new element that continues to separate the factions is the use of complete commodity workstation nodes, as in the SP-1, SP-2 and various workstation clusters using merging high bandwidth networks, such as ATM.[And*94,Kun*91,Pfi95]. The costs of weaker integration into the memory system, imperfect network reliability, and general purpose system requirements have tended to keep these systems more closely aligned with traditional message passing, although the future developments are far from clear.

1.7 Exercises

1.1 Compute the annual growth rate in number of transistors, die size, and clock rate by fitting an exponential to the technology leaders using available data inTable 1-1.

1.2 Compute the annual performance growth rates for each of the benchmarks shown in Table 1-1. Comment on the differences that you observe

Table 1-1 Performance of leading workstations

Year	Machine	SpecInt	SpecFP	Linpack	n=1000	Peak FP
Sun 4/260	1987	9	6	1.1	1.1	3.3
MIPS M/120	1988	13	10.2	2.1	4.8	6.7
MIPS M/2000	1989	18	21	3.9	7.9	10
IBM RS6000/540	1990	24	44	19	50	60
HP 9000/750	1991	51	101	24	47	66
DEC Alpha AXP	1992	80	180	30	107	150
DEC 7000/610	1993	132.6	200.1	44	156	200
AlphaServer 2100	1994	200	291	43	129	190

Generally, in evaluating performance trade-offs we will evaluate the improvement in performance, or speedup, due to some enhancement. Formally,

$$Speedup \text{ due to enhancement E} = \frac{Time_{without\ E}}{Time_{with\ E}} = \frac{Performance_{with\ E}}{Performance_{without\ E}}$$

In particular, we will often refer to the speedup as a function of the machine parallel, e.g., the number of processors.

1.3 Suppose you are given a program which does a fixed amount of work and some fraction s of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on P processors. Assuming T_1 is the time taken on one processor, derive a formula for T_p the time taken on P processors. Use this to get a formula giving an upper bound on the potential speedup on P processors. (This is a variant of what is often called Amdahl's Law[Amd67].) Explain why it is an upper bound?

1.4 Given a histogram of available parallelism such as that shown in Figure 1-7, where f_i is the fraction of cycles on an ideal machine in which i instructions issue, derive a generalization of Amdahl's law to estimate the potential speedup on a k-issue superscalar machine. Apply your formula the histogram data in Figure 1-7 to produce the speedup curve shown in that figure.

1.5 Locate the current TPC performance data on the web and compare the mix of system configurations, performance, and speedups obtained on those machines with the data presented in Figure 1-4.

Programming Models

1.6 In message passing models each process is provided with a special variable or function that gives its unique number or rank among the set of processes executing a program. Most shared-memory programming systems provide a fetch&inc operation, which reads the value of a location and atomically increments the location. Write a little pseudo-code to show how

to use fetch&add to assign each process a unique number. Can you determine the number of processes comprising a shared memory parallel program in a similar way?

1.7 To move an n-byte message along H links in an unloaded store-and-forward network takes time $H\frac{n}{W} + (H-1)R$, where W is the raw link bandwidth, and R is the routing delay per hop. In a network with cut-through routing this takes time $\frac{n}{W} + (H-1)R$. Consider an 8x8 grid consisting of 40 MB/s links and routers with 250ns of delay. What is the minimum, maximum, and average time to move a 64 byte message through the network? A 246 byte message?

1.8 Consider a simple 2D finite difference scheme where at each step every point in the matrix updated by a weighted average of its four neighbors,

$$A[i, j] = A[i, j] - w(A[i-1, j] + A[i+1, j] + A[i, j-1] + A[i, j+1])$$

All the values are 64-bit floating point numbers. Assuming one element per processor and 1024x1024 elements, how much data must be communicated per step? Explain how this computation could be mapped onto 64 processors so as to minimize the data traffic. Compute how much data must be communicated per step.

Latency and bandwidth

1.9 Consider the simple pipelined component described in Example 1-2. Suppose that the application alternates beween bursts of m independent operations on the component and phases of computation lasting T ns that do not use the component. Develop an expression describing the execution time of the program based on these parameters. Compare this with the unpipelined and fully pipelined bounds. At what points to you get the maximum discrepancy between the models? How large is it as a fraction of overal execution time?

1.10 Show that Equation 1.4 follows from Equation 1.3.

1.11 What is the x-intercept of the line in Equation 1.3?

If we consider loading a cache line from memory the transfer time is the time to actually transmit the data across the bus. The start-up includes the time to obtain access to the bus, convey the address, access the memory, and possibly to place the data in the cache before responding to the processor. However, in a modern processor with dynamic instruction scheduling, the overhead may include only the portion spent accessing the cache to detect the miss and placing the request on the bus. The memory access portion contributes to latency, which can potentially be hidden by the overlap with execution of instructions that do not depend on the result of the load.

1.12 Suppose we have a machine with a 64-bit wide bus running at 40 MHz. It takes 2 bus cycles to arbitrate for the bus and present the address. The cache line size is 32 bytes and the memory access time is 100ns. What is the latency for a read miss? What bandwidth is obtained on this transfer?

1.13 Suppose this 32-byte line is transferred to another processor and the communication architecture imposes a start-up cost of 2μs and data transfer bandwidth of 20MB/s. What is the total latency of the remote operation?

If we consider sending an n-byte message to another processor, we may use the same model. The start-up can be thought of as the time for a zero length message; it includes the software overhead

on the two processors, the cost of accessing the network interface, and the time to actually cross the network. The transfer time is usually determined by the point along the path with the least bandwidth, i.e., the bottleneck.

1.14 Suppose we have a machine with a message start-up of 100μs and a asymptotic peak bandwidth of 80MB/s. At what size message is half of the peak bandwidth obtained?

1.15 Derive a general formula for the "half-power point" in terms of the start-up cost and peak bandwidth.

The model makes certain basic trade-offs clear. For example, longer transfers take more time, but obtain higher bandwidth because the start-up cost is amortized over more data movement. This observation can help guide design trade-offs, at least up to a point where the collection of data transfers interact to increase the start-up cost or reduce the effective bandwidth.

In some cases we will use the model in Equation 1.6 for estimating data transfer performance based on design parameters, as in the examples above. In other cases, we will use it as an empirical tool and fit measurements to a line to determine the effective start-up and peak bandwidth of a portion of a system. Observe, for example, that if data undergoes a series of copies as part of a transfer

1.16 Assuming that before transmitting a message the data must be copied into a buffer. The basic message time is as in Exercise 1.14, but the copy is performed at a cost of 5 cycles per 32-bit words on a 100 MHz machine. Given an equation for the expected user-level message time. How does the cost of a copy compare with a fixed cost of, say, entering the operating system.

1.17 Consider a machine running at 100 MIPS on some workload with the following mix: 50% ALU, 20% loads, 10% stores, 10% branches. Suppose the instruction miss rate is 1%, the data miss rate is 5%, the cache line size is 32 bytes. For the purpose of this calculation, treat a store miss as requiring two cache line transfers, one to load the newly update line and one to replace the dirty line. If the machine provides a 250 MB/s bus, how many processors can it accommodate at 50% of peak bus bandwidth? What is the bandwidth demand of each processor?

The scenario is Exercise 1.17 is a little rosy because it looks only at the sum of the average bandwidths, which is why we left 50% headroom on the bus. In fact, what happens as the bus approaches saturation is that it takes longer to obtain access for the bus, so it looks to the processor as if the memory system is slower. The effect is to slow down all of the processors in the system, thereby reducing their bandwidth demand. Let's try a analogous calculation from the other direction.

1.18 Assume the instruction mix and miss rate as in Exercise 1.17, but ignore the MIPS, since that depends on the performance of the memory system. Assume instead that the processor run as 100 MHz and has an ideal CPI (with an perfect memory system). The unloaded cache miss penalty is 20 cycles. You can ignore the write-back for stores. (As a starter, you might want to compute the MIPS rate for this new machine.) Assume that the memory system, i.e., the bus and the memory controller is utilized throughout the miss. What is the utilization of the memory system, U_1, with a single processor? From this result, estimate the number of processors that could be supported before the processor demand would exceed the available bus bandwidth.

1.19 Of course, no matter how many processors you place on the bus, they will never exceed the available bandwidth. Explains what happens to processor performance in response to bus contention. Can you formalize your observations?

CHAPTER 2   # Parallel Programs

2.1 Introduction

To understand and evaluate design decisions in a parallel machine, we must have an idea of the software that runs on the machine. Understanding program behavior led to some of the most important advances in uniprocessors, including memory hierarchies and instruction set design. It is all the more important in multiprocessors, both because of the increase in degrees of freedom and because of the much greater performance penalties caused by to mismatches between applications and systems.

Understanding parallel software is important for algorithm designers, for programmers, and for architects. As algorithm designers, it helps us focus on designing algorithms that can be run effectively in parallel on real systems. As programmers, it helps us understand the key performance issues and obtain the best performance from a system. And as architects, it helps us under-

stand the workloads we are designing against and their important degrees of freedom. Parallel software and its implications will be the focus of the next three chapters of this book. This chapter describes the process of creating parallel programs in the major programming models. The next chapter focuses on the performance issues that must be addressed in this process, exploring some of the key interactions between parallel applications and architectures. And the following chapter relies on this understanding of software and interactions to develop guidelines for using parallel workloads to evaluate architectural tradeoffs. In addition to architects, the material in these chapters is useful for users of parallel machines as well: the first two chapters for programmers and algorithm designers, and the third for users making decisions about what machines to procure. However, the major focus is on issues that architects should understand before they get into the nuts and bolts of machine design and architectural tradeoffs, so let us look at it from this perspective.

As architects of sequential machines, we generally take programs for granted: The field is mature and there is a large base of programs than can be (or must be) viewed as fixed. We optimize the machine design against the requirements of these programs. Although we recognize that programmers may further optimize their code as caches become larger or floating-point support is improved, we usually evaluate new designs without anticipating such software changes. Compilers may evolve along with the architecture, but the source program is still treated as fixed. In parallel architecture, there is a much stronger and more dynamic interaction between the evolution of machine designs and that of parallel software. Since parallel computing is all about performance, programming tends to be oriented towards taking advantage of what machines provide. Parallelism offers a new degree of freedom—the number of processors—and higher costs for data access and coordination, giving the programmer a wide scope for software optimizations. Even as architects, we therefore need to open up the application "black box". Understanding the important aspects of the process of creating parallel software, the focus of this chapter, helps us appreciate the role and limitations of the architecture. The deeper look at performance issues in the next chapter will shed greater light on hardware-software tradeoffs.

Even after a problem and a good sequential algorithm for it are determined, there is a substantial process involved in arriving at a parallel program and the execution characteristics that it offers to a multiprocessor architecture. This chapter presents general principles of the parallelization process, and illustrates them with real examples. The chapter begins by introducing four actual problems that serve as case studies throughout the next two chapters. Then, it describes the four major steps in creating a parallel program—using the case studies to illustrate—followed by examples of how a simple parallel program might be written in each of the major programming models. As discussed in Chapter 1, the dominant models from a programming perspective narrow down to three: the data parallel model, a shared address space, and message passing between private address spaces. This chapter illustrates the primitives provided by these models and how they might be used, but is not concerned much with performance. After the performance issues in the parallelization process are understood in the next chapter, the four application case studies will be treated in more detail to create high-performance versions of them.

2.2 Parallel Application Case Studies

We saw in the previous chapter that multiprocessors are used for a wide range of applications—from multiprogramming and commercial computing to so-called "grand challenge" scientific problems—and that the most demanding of these applications for high-end systems tend to be

from scientific and engineering computing. Of the four case studies we refer to throughout this chapter and the next, two are from scientific computing, one is from commercial computing and one from computer graphics. Besides being from different application domains, the case studies are chosen to represent a range of important behaviors found in other parallel programs as well.

Of the two scientific applications, one simulates the motion of ocean currents by discretizing the problem on a set of regular grids and solving a system of equations on the grids. This technique of discretizing equations on grids is very common in scientific computing, and leads to a set of very common communication patterns. The second case study represents another major form of scientific computing, in which rather than discretizing the domain on a grid the computational domain is represented as a large number of bodies that interact with one another and move around as a result of these interactions. These so-called N-body problems are common in many areas such as simulating galaxies in astrophysics (our specific case study), simulating proteins and other molecules in chemistry and biology, and simulating electromagnetic interactions. As in many other areas, hierarchical algorithms for solving these problems have become very popular. Hierarchical N-body algorithms, such as the one in our case study, have also been used to solve important problems in computer graphics and some particularly difficult types of equation systems. Unlike the first case study, this one leads to irregular, long-range and unpredictable communication.

The third case study is from computer graphics, a very important consumer of moderate-scale multiprocessors. It traverses a three-dimensional scene and highly irregular and unpredictable ways, and renders it into a two-dimensional image for display. The last case study represents the increasingly important class of commercial applications that analyze the huge volumes of data being produced by our information society to discover useful knowledge, categories and trends. These information processing applications tend to be I/O intensive, so parallelizing the I/O activity effectively is very important. The first three case studies are part of a benchmark suite [SWG92] that is widely used in architectural evaluations in the literature, so there is a wealth of detailed information available about them. They will be used to illustrate architectural tradeoffs in this book as well.

2.2.1 Simulating Ocean Currents

To model the climate of the earth, it is important to understand how the atmosphere interacts with the oceans that occupy three fourths of the earth's surface. This case study simulates the motion of water currents in the ocean. These currents develop and evolve under the influence of several physical forces, including atmospheric effects, wind, and friction with the ocean floor. Near the ocean walls there is additional "vertical" friction as well, which leads to the development of eddy currents. The goal of this particular application case study is to simulate these eddy currents over time, and understand their interactions with the mean ocean flow.

Good models for ocean behavior are complicated: Predicting the state of the ocean at any instant requires the solution of complex systems of equations, which can only be performed numerically by computer. We are, however, interested in the behavior of the currents over time. The actual physical problem is continuous in both space (the ocean basin) and time, but to enable computer simulation we discretize it along both dimensions. To discretize space, we model the ocean basin as a grid of equally spaced points. Every important variable—such as pressure, velocity, and various currents—has a value at each grid point in this discretization. This particular application uses not a three-dimensional grid but a set of two-dimensional, horizontal cross-sections through

the ocean basin, each represented by a two-dimensional grid of points (see Figure 2-1). For sim-

| (a) Cross-sections | (b) Spatial discretization of a cross-section |
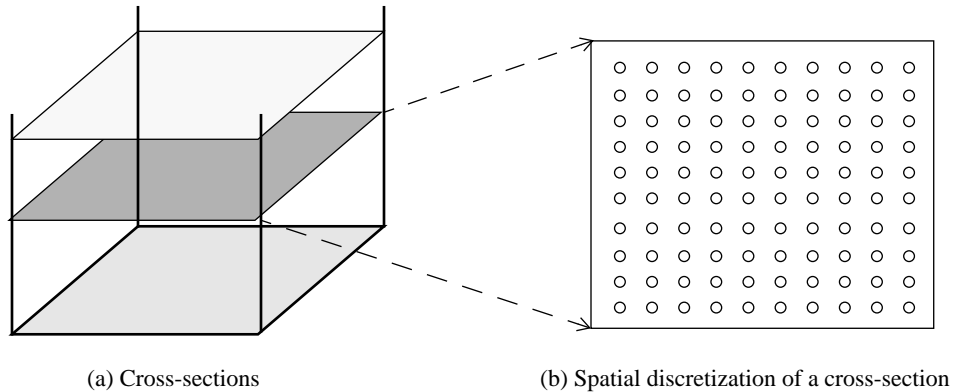
Figure 2-1 Horizontal cross-sections through an ocean basin, and their spatial discretization into regular grids.

plicity, the ocean is modeled as a rectangular basin and the grid points are assumed to be equally spaced. Each variable is therefore represented by a separate two-dimensional array for each cross-section through the ocean. For the time dimension, we discretize time into a series of finite time-steps. The equations of motion are solved at all the grid points in one time-step, the state of the variables is updated as a result, and the equations of motion are solved again for the next time-step, and so on repeatedly.

Every time-step itself consists of several computational phases. Many of these are used to set up values for the different variables at all the grid points using the results from the previous time-step. Then there are phases in which the system of equations governing the ocean circulation are actually solved. All the phases, including the solver, involve sweeping through all points of the relevant arrays and manipulating their values. The solver phases are somewhat more complex, as we shall see when we discuss this case study in more detail in the next chapter.

The more grid points we use in each dimension to represent our fixed-size ocean, the finer the spatial resolution of our discretization and the more accurate our simulation. For an ocean such as the Atlantic, with its roughly 2000km-x-2000km span, using a grid of 100-x-100 points implies a distance of 20km between points in each dimension. This is not a very fine resolution, so we would like to use many more grid points. Similarly, shorter physical intervals between time-steps lead to greater simulation accuracy. For example, to simulate five years of ocean movement updating the state every eight hours we would need about 5500 time-steps.

The computational demands for high accuracy are large, and the need for multiprocessing is clear. Fortunately, the application also naturally affords a lot of concurrency: many of the set-up phases in a time-step are independent of one another and therefore can be done in parallel, and the processing of different grid points in each phase or grid computation can itself be done in parallel. For example, we might assign different parts of each ocean cross-section to different processors, and have the processors perform their parts of each phase of computation (a data-parallel formulation).

2.2.2 Simulating the Evolution of Galaxies

Our second case study is also from scientific computing. It seeks to understand the evolution of stars in a system of galaxies over time. For example, we may want to study what happens when galaxies collide, or how a random collection of stars folds into a defined galactic shape. This problem involves simulating the motion of a number of bodies (here stars) moving under forces exerted on each by all the others, an *n-body* problem. The computation is discretized in space by treating each star as a separate body, or by sampling to use one body to represent many stars. Here again, we discretize the computation in time and simulate the motion of the galaxies for many time-steps. In each time-step, we compute the gravitational forces exerted on each star by all the others and update the position, velocity and other attributes of that star.

Computing the forces among stars is the most expensive part of a time-step. A simple method to compute forces is to calculate pairwise interactions among all stars. This has $O(n^2)$ computational complexity for *n* stars, and is therefore prohibitive for the millions of stars that we would like to simulate. However, by taking advantage of insights into the force laws, smarter hierarchical algorithms are able to reduce the complexity to $O(n \log n)$. This makes it feasible to simulate problems with millions of stars in reasonable time, but only by using powerful multiprocessors. The basic insight that the hierarchical algorithms use is that since the strength of the gravitational interaction falls off with distance as $G\dfrac{m_1 m_2}{r^2}$, the influences of stars that are further away are weaker and therefore do not need to be computed as accurately as those of stars that are close by. Thus, if a group of stars is far enough away from a given star, then their effect on the star does not have to be computed individually; as far as that star is concerned, they can be approximated as a single star at their center of mass without much loss in accuracy (Figure 2-2). The further away
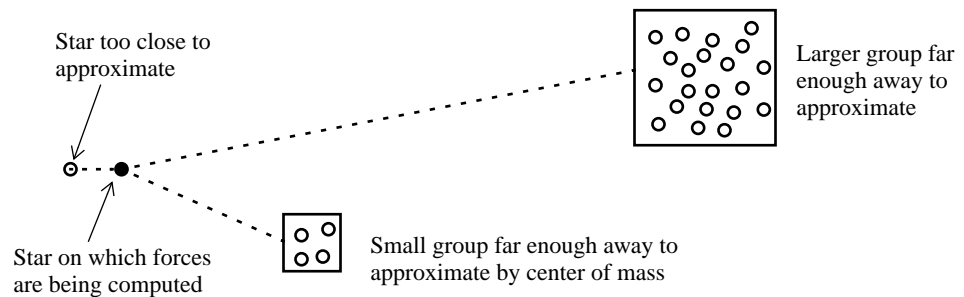
Figure 2-2 The insight used by hierarchical methods for n-body problems.

A group of bodies that is far enough away from a given body may be approximated by the center of mass of the group. The further apart the bodies, the larger the group that may be thus approximated.

the stars from a given star, the larger the group that can be thus approximated. In fact, the strength of many physical interactions falls off with distance, so hierarchical methods are becoming increasingly popular in many areas of computing.

The particular hierarchical force-calculation algorithm used in our case study is the Barnes-Hut algorithm. The case study is called Barnes-Hut in the literature, and we shall use this name for it as well. We shall see how the algorithm works in Section 3.6.2. Since galaxies are denser in some regions and sparser in others, the distribution of stars in space is highly irregular. The distribution

also changes with time as the galaxy evolves. The nature of the hierarchy implies that stars in denser regions interact with more other stars and centers of mass—and hence have more work associated with them—than stars in sparser regions. There is ample concurrency across stars within a time-step, but given the irregular and dynamically changing nature the challenge is to exploit it efficiently on a parallel architecture.

2.2.3 Visualizing Complex Scenes using Ray Tracing

Our third case study is the visualization of complex scenes in computer graphics. A common technique used to render such scenes into images is ray tracing. The scene is represented as a set of objects in three-dimensional space, and the image being rendered is represented as a two-dimensional array of pixels (picture elements) whose color, opacity and brightness values are to be computed. The pixels taken together represent the image, and the resolution of the image is determined by the distance between pixels in each dimension. The scene is rendered as seen from a specific viewpoint or position of the eye. Rays are shot from that viewpoint through every pixel in the image plane and into the scene. The algorithm traces the paths of these rays—computing their reflection, refraction, and lighting interactions as they strike and reflect off objects—and thus computes values for the color and brightness of the corresponding pixels. There is obvious parallelism across the rays shot through different pixels. This application will be referred to as Raytrace.

2.2.4 Mining Data for Associations

Information processing is rapidly becoming a major marketplace for parallel systems. Businesses are acquiring a lot of data about customers and products, and devoting a lot of computational power to automatically extracting useful information or "knowledge" from these data. Examples from a customer database might include determining the buying patterns of demographic groups or segmenting customers according to relationships in their buying patterns. This process is called data mining. It differs from standard database queries in that its goal is to identify implicit trends and segmentations, rather than simply look up the data requested by a direct, explicit query. For example, finding all customers who have bought cat food in the last week is not data mining; however, segmenting customers according to relationships in their age group, their monthly income, and their preferences in pet food, cars and kitchen utensils is.

A particular, quite directed type of data mining is mining for associations. Here, the goal is to discover relationships (associations) among the information related to different customers and their transactions, and to generate rules for the inference of customer behavior. For example, the database may store for every transaction the list of items purchased in that transaction. The goal of the mining may be to determine associations between sets of commonly purchased items that tend to be purchased together; for example, the conditional probability $P(S_1|S_2)$ that a certain set of items S_1 is found in a transaction given that a different set of items S_2 is found in that transaction, where S_1 and S_2 are sets of items that occur often in transactions.

Consider the problem a little more concretely. We are given a database in which the records correspond to customer purchase transactions, as described above. Each transaction has a transaction identifier and a set of attributes or items, for example the items purchased. The first goal in mining for associations is to examine the database and determine which sets of k items, say, are found to occur together in more than a given threshold fraction of the transactions. A set of items (of any size) that occur together in a transaction is called an *itemset*, and an itemset that is found

in more than that threshold percentage of transactions is called a *large itemset*. Once the large itemsets of size *k* are found—together with their frequencies of occurrence in the database of transactions—determining the association rules among them is quite easy. The problem we consider therefore focuses on discovering the large itemsets of size *k* and their frequencies.

The data in the database may be in main memory, or more commonly on disk. A simple way to solve the problem is to first determine the large itemsets of size one. From these, a set of candidate itemsets of size two items can be constructed—using the basic insight that if an itemset is large then all its subsets must also be large—and their frequency of occurrence in the transaction database counted. This results in a list of large itemsets of size two. The process is repeated until we obtain the large itemsets of size *k*. There is concurrency in examining large itemsets of size *k-1* to determine candidate itemsets of size *k*, and in counting the number of transactions in the database that contain each of the candidate itemsets.

2.3 The Parallelization Process

The four case studies—Ocean, Barnes-Hut, Raytrace and Data Mining—offer abundant concurrency, and will help illustrate the process of creating effective parallel programs in this chapter and the next. For concreteness, we will assume that the sequential algorithm that we are to make parallel is given to us, perhaps as a description or as a sequential program. In many cases, as in these case studies, the best sequential algorithm for a problem lends itself easily to parallelization; in others, it may not afford enough parallelism and a fundamentally different algorithm may be required. The rich field of parallel algorithm design is outside the scope of this book. However, there is in all cases a significant process of creating a good parallel program that implements the chosen (sequential) algorithm, and we must understand this process in order to program parallel machines effectively and evaluate architectures against parallel programs.

At a high level, the job of parallelization involves identifying the work that can be done in parallel, determining how to distribute the work and perhaps the data among the processing nodes, and managing the necessary data access, communication and synchronization. Note that work includes computation, data access, and input/output activity. The goal is to obtain high performance while keeping programming effort and the resource requirements of the program low. In particular, we would like to obtain good speedup over the best sequential program that solves the same problem. This requires that we ensure a balanced distribution of work among processors, reduce the amount of interprocessor communication which is expensive, and keep the overheads of communication, synchronization and parallelism management low.

The steps in the process of creating a parallel program may be performed either by the programmer or by one of the many layers of system software that intervene between the programmer and the architecture. These layers include the compiler, runtime system, and operating system. In a perfect world, system software would allow users to write programs in the form they find most convenient (for example, as sequential programs in a high-level language or as an even higher-level specification of the problem), and would automatically perform the transformation into efficient parallel programs and executions. While much research is being conducted in parallelizing compiler technology and in programming languages that make this easier for compiler and runtime systems, the goals are very ambitious and have not yet been achieved. In practice today, the vast majority of the process is still the responsibility of the programmer, with perhaps some help from the compiler and runtime system. Regardless of how the responsibility is divided among

these parallelizing agents, the issues and tradeoffs are similar and it is important that we understand them. For concreteness, we shall assume for the most part that the programmer has to make all the decisions.

Let us now examine the parallelization process in a more structured way, by looking at the actual steps in it. Each step will address a subset of the issues needed to obtain good performance. These issues will be discussed in detail in the next chapter, and only mentioned briefly here.

2.3.1 Steps in the Process

To understand the steps in creating a parallel program, let us first define three few important concepts: tasks, processes and processors. A *task* is an arbitrarily defined piece of the work done by the program. It is the smallest unit of concurrency that the parallel program can exploit; i.e., an individual task is executed by only one processor, and concurrency is exploited across tasks. In the Ocean application we can think of a single grid point in each phase of computation as being a task, or a row of grid points, or any arbitrary subset of a grid. We could even consider an entire grid computation to be a single task, in which case parallelism is exploited only across independent grid computations. In Barnes-Hut a task may be a particle, in Raytrace a ray or a group of rays, and in Data Mining it may be checking a single transaction for the occurrence of an itemset. What exactly constitutes a task is not prescribed by the underlying sequential program; it is a choice of the parallelizing agent, though it usually matches some natural granularity of work in the sequential program structure. If the amount of work a task performs is small, it is called a *fine-grained* task; otherwise, it is called *coarse-grained*.

A *process* (referred to interchangeably hereafter as a *thread*) is an abstract entity that performs tasks.[1] A parallel program is composed of multiple cooperating processes, each of which performs a subset of the tasks in the program. Tasks are assigned to processes by some *assignment* mechanism. For example, if the computation for each row in a grid in Ocean is viewed as a task, then a simple assignment mechanism may be to give an equal number of adjacent rows to each process, thus dividing the ocean cross section into as many horizontal slices as there are processes. In data mining, the assignment may be determined by which portions of the database are assigned to each process, and by how the itemsets within a candidate list are assigned to processes to look up the database. Processes may need to communicate and synchronize with one another to perform their assigned tasks. Finally, the way processes perform their assigned tasks is by executing them on the physical *processors* in the machine.

It is important to understand the difference between processes and processors from a parallelization perspective. While processors are physical resources, processes provide a convenient way of abstracting or *virtualizing* a multiprocessor: We initially write parallel programs in terms of processes not physical processors; mapping processes to processors is a subsequent step. The number of processes does not *have* to be the same as the number of processors available to the program. If there are more processes, they are multiplexed onto the available processors; if there are fewer processes, then some processors will remain idle.

---

1. In Chapter 1 we used the correct operating systems definition of a process: an address space and one or more threads of control that share that address space. Thus, processes and threads are distinguished in that definition. To simplify our discussion of parallel programming in this chapter, we do not make this distinction but assume that a process has only one thread of control.
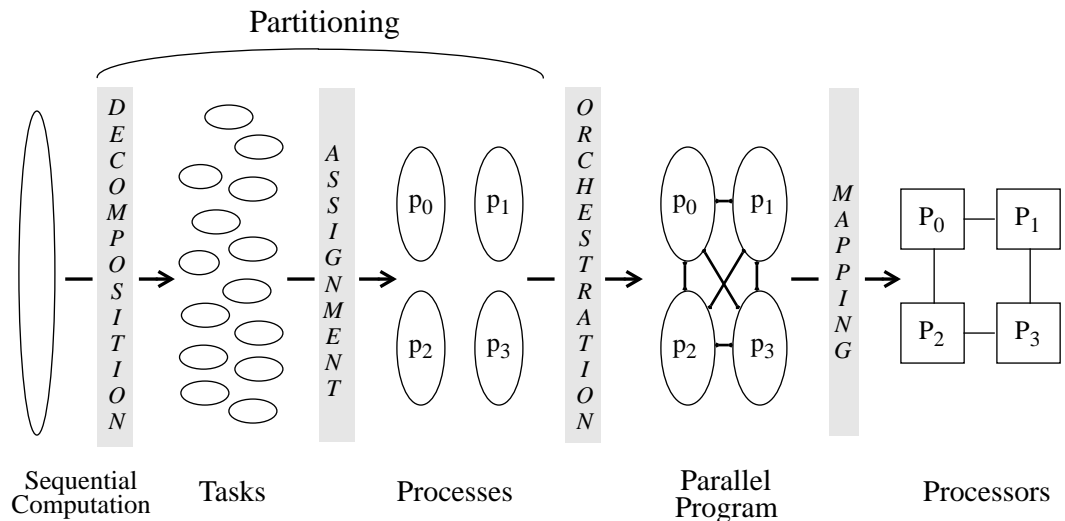
Figure 2-3 Step in parallelization, and the relationships among tasks, processes and processors.

The decomposition and assignment phases are together called partitioning. The orchestration phase coordinates data access, communication and synchronization among processes, and the mapping phase maps them to physical processors.

Given these concepts, the job of creating a parallel program from a sequential one consists of four steps, illustrated in Figure 2-3:

1. **Decomposition** of the computation into tasks,

2. **Assignment** of tasks to processes,

3. **Orchestration** of the necessary data access, communication and synchronization among processes, and

4. **Mapping** or binding of processes to processors.

Together, decomposition and assignment are called partitioning, since they divide the work done by the program among the cooperating processes. Let us examine the steps and their individual goals a little further.

Decomposition

Decomposition means breaking up the computation into a collection of tasks. For example, tracing a single ray in Raytrace may be a task, or performing a particular computation on an individual grid point in Ocean. In general, tasks may become available dynamically as the program executes, and the number of tasks available at a time may vary over the execution of the program. The maximum number of tasks available at a time provides an upper bound on the number of processes (and hence processors) that can be used effectively at that time. Hence, the major goal in decomposition is to *expose enough concurrency* to keep the processes busy at all times, yet not so much that the overhead of managing the tasks becomes substantial compared to the useful work done.

Limited concurrency is the most fundamental limitation on the speedup achievable through parallelism, not just the fundamental concurrency in the underlying problem but also how much of this concurrency is exposed in the decomposition. The impact of available concurrency is codified in one of the few "laws" of parallel computing, called *Amdahl's Law*. If some portions of a program's execution don't have as much concurrency as the number of processors used, then some processors will have to be idle for those portions and speedup will be suboptimal. To see this in its simplest form, consider what happens if a fraction s of a program's execution time on a uniprocessor is inherently sequential; that is, it cannot be parallelized. Even if the rest of the program is parallelized to run on a large number of processors in infinitesimal time, this sequential time will remain. The overall execution time of the parallel program will be at least s, normalized to a total sequential time of 1, and the speedup limited to $1/s$. For example, if s=0.2 (20% of the program's execution is sequential), the maximum speedup available is 1/0.2 or 5 regardless of the number of processors used, even if we ignore all other sources of overhead.

Example 2-1   Consider a simple example program with two phases. In the first phase, a single operation is performed independently on all points of a two-dimensional n-by-n grid, as in Ocean. In the second, the sum of the n^2 grid point values is computed. If we have p processors, we can assign n^2/p points to each processor and complete the first phase in parallel in time n^2/p. In the second phase, each processor can add each of its assigned n^2/p values into a global sum variable. What is the problem with this assignment, and how can we expose more concurrency?

Answer   The problem is that the accumulations into the global sum must be done one at a time, or *serialized*, to avoid corrupting the sum value by having two processors try to modify it simultaneously (see mutual exclusion in Section 2.4.5). Thus, the second phase is effectively serial and takes n^2 time regardless of p. The total time in parallel is $n^2/p + n^2$, compared to a sequential time of $2n^2$, so the speedup is at most

$\dfrac{2n^2}{\dfrac{n^2}{p} + n^2}$ or $\dfrac{2p}{p+1}$, which is at best 2 even if a very large number of processors is used.

We can expose more concurrency by using a little trick. Instead of summing each value directly into the global sum, serializing all the summing, we divide the second phase into two phases. In the new second phase, a process sums its assigned values independently into a private sum. Then, in the third phase, processes sum their private sums into the global sum. The second phase is now fully parallel; the third phase is serialized as before, but there are only p operations in it, not n. The

total parallel time is $n^2/p + n^2/p + p$, and the speedup is at best $p \times \dfrac{2n^2}{2n^2 + p^2}$. If n is

large relative to p, this speedup limit is almost linear in the number of processors used. Figure 2-4 illustrates the improvement and the impact of limited concurrency.
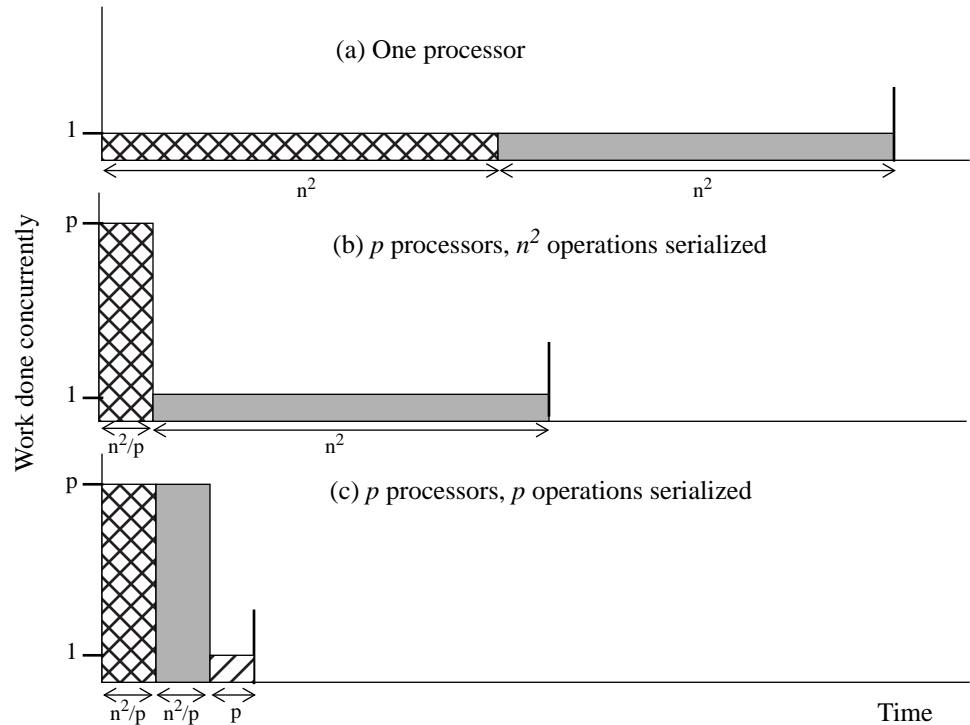
Figure 2-4 Illustration of the impact of limited concurrency.

The x-axis is time, and the y-axis is the amount of work available (exposed by the decomposition) to be done in parallel at a given time. (a) shows the profile for a single processor. (b) shows the original case in the example, which is divided into two phases: one fully concurrent, and one fully serialized. (c) shows the improved version, which is divided into three phases: the first two fully concurrent, and the last fully serialized but with a lot less work in it ($O(p)$ rather than $O(n)$).

More generally, given a decomposition and a problem size, we can construct a *concurrency profile* which depicts how many operations (or tasks) are available to be performed concurrently in the application at a given time. The concurrency profile is a function of the problem, the decomposition and the problem size but is independent of the number of processors, effectively assuming that an infinite number of processors is available. It is also independent of the assignment or orchestration. These concurrency profiles may be easy to analyze (as we shall see for matrix factorization in Exercise 2.4) or they may be quite irregular. For example, Figure 2-5 shows a concurrency profile of a parallel event-driven simulation for the synthesis of digital logic systems. The X-axis is time, measured in clock cycles of the circuit being simulated. The Y-axis or amount of concurrency is the number of logic gates in the circuit that are ready to be evaluated at a given time, which is a function of the circuit, the values of its inputs, and time. There is a wide range of unpredictable concurrency across clock cycles, and some cycles with almost no concurrency.

The area under the curve in the concurrency profile is the total amount of work done, i.e. the number of operations or tasks computed, or the "time" taken on a single processor. It's horizontal extent is a lower bound on the "time" that it would take to run the best parallel program given that decomposition, assuming an infinitely large number of processors. The area divided by the hori-
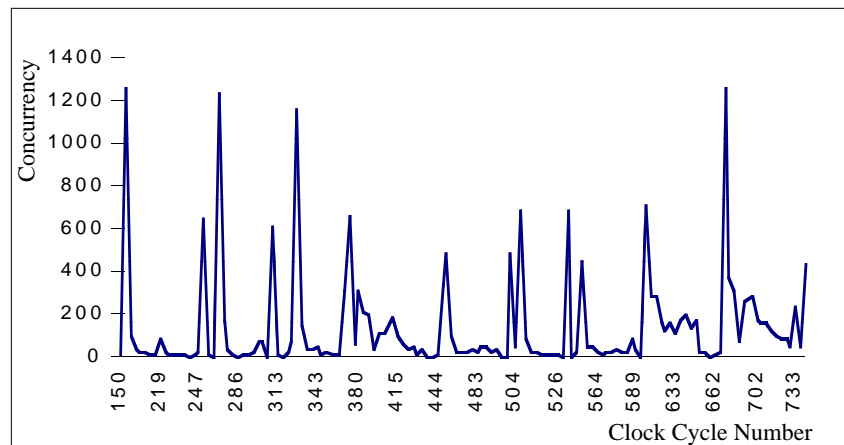
Figure 2-5 Concurrency profile for a distributed-time, discrete-event logic simulator.

The circuit being simulated is a simple MIPS R6000 microprocessor. The y-axis shows the number of logic elements available for evaluation in a given clock cycle.

zontal extent therefore gives us a limit on the achievable speedup with unlimited number of processors, which is thus simply the average concurrency available in the application over time. A rewording of Amdahl's law may therefore be:

$$Speedup \leq \frac{AreaUnderConcurencyProfile}{HorizontalExtentofConcurrencyProfile}$$

Thus, if f_k be the number of X-axis points in the concurrency profile that have concurrency k, then we can write Amdahl's Law as:

$$Speedup(p) \leq \frac{\displaystyle\sum_{k=1}^{\infty} f_k k}{\displaystyle\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil} \ . \qquad \textbf{(EQ 2.1)}$$

It is easy to see that if the total work $\displaystyle\sum_{k=1}^{\infty} f_k k$ is normalized to 1 and a fraction s of this is serial, then the speedup with an infinite number of processors is limited by $\frac{1}{s}$, and that with p processors is limited by $\dfrac{1}{s + \dfrac{1-s}{p}}$. In fact, Amdahl's law can be applied to any overhead of parallelism, not just limited concurrency, that is not alleviated by using more processors. For now, it quantifies the importance of exposing enough concurrency as a first step in creating a parallel program.

Assignment

Assignment means specifying the mechanism by which tasks will be distributed among processes. For example, which process is responsible for computing forces on which stars in Barnes-Hut, and which process will count occurrences of which itemsets and in which parts of the database in Data Mining? The primary performance goals of assignment are to *balance the workload* among processes, to *reduce interprocess communication*, and to *reduce the runtime overheads of managing the assignment*. Balancing the workload is often referred to as *load balancing*. The workload to be balanced includes computation, input/output and data access or communication, and programs that are not balance these well among processes are said to be load imbalanced.

Achieving these performance goals simultaneously can appear intimidating. However, most programs lend themselves to a fairly structured approach to partitioning (decomposition and assignment). For example, programs are often structured in phases, and candidate tasks for decomposition within a phase are often easily identified as seen in the case studies. The appropriate assignment of tasks is often discernible either by inspection of the code or from a higher-level understanding of the application. And where this is not so, well-known heuristic techniques are often applicable. If the assignment is completely determined at the beginning of the program—or just after reading and analyzing the input—and does not change thereafter, it is called a *static or predetermined assignment*; if the assignment of work to processes is determined at runtime as the program executes—perhaps to react to load imbalances—it is called a *dynamic assignment*. We shall see examples of both. Note that this use of static is a little different than the "compile-time" meaning typically used in computer science. Compile-time assignment that does not change at runtime would indeed be static, but the term is more general here.

Decomposition and assignment are the major algorithmic steps in parallelization. They are *usually* independent of the underlying architecture and programming model, although sometimes the cost and complexity of using certain primitives on a system can impact decomposition and assignment decisions. As architects, we assume that the programs that will run on our machines are reasonably partitioned. There is nothing we can do if a computation is not parallel enough or not balanced across processes, and little we may be able to do if it overwhelms the machine with communication. As programmers, we usually focus on decomposition and assignment first, independent of the programming model or architecture, though in some cases the properties of the latter may cause us to revisit our partitioning strategy.

Orchestration

This is the step where the architecture and programming model play a large role, as well as the programming language itself. To execute their assigned tasks, processes need mechanisms to name and access data, to exchange data (communicate) with other processes, and to synchronize with one another. Orchestration uses the available mechanisms to accomplish these goals correctly and efficiently. The choices made in orchestration are much more dependent on the programming model, and on the efficiencies with which the primitives of the programming model and communication abstraction are supported, than the choices made in the previous steps. Some questions in orchestration include how to organize data structures and schedule tasks to exploit locality, whether to communicate implicitly or explicitly and in small or large messages, and how exactly to organize and express interprocess communication and synchronization. Orchestration also includes scheduling the tasks assigned to a process temporally, i.e. deciding the order in which they are executed. The programming language is important both because this is the step in

which the program is actually written and because some of the above tradeoffs in orchestration are influenced strongly by available language mechanisms and their costs.

The major performance goals in orchestration are *reducing the cost of the communication and synchronization* as seen by the processors, *preserving locality of data reference*, *scheduling tasks* so that those on which many other tasks depend are completed early, and *reducing the overheads of parallelism management*. The job of architects is to provide the appropriate primitives with efficiencies that simplify successful orchestration. We shall discuss the major aspects of orchestration further when we see how programs are actually written.

Mapping

The cooperating processes that result from the decomposition, assignment and orchestration steps constitute a full-fledged parallel program on modern systems. The program may control the mapping of processes to processors, but if not the operating system will take care of it, providing a parallel execution. Mapping tends to be fairly specific to the system or programming environment. In the simplest case, the processors in the machine are partitioned into fixed subsets, possibly the entire machine, and only a single program runs at a time in a subset. This is called *space-sharing*. The program can bind or *pin* processes to processors to ensure that they do not migrate during the execution, or can even control exactly which processor a process runs on so as to preserve locality of communication in the network topology. Strict space-sharing schemes, together with some simple mechanisms for time-sharing a subset among multiple applications, have so far been typical of large-scale multiprocessors.

At the other extreme, the operating system may dynamically control which process runs where and when—without allowing the user any control over the mapping—to achieve better resource sharing and utilization. Each processor may use the usual multiprogrammed scheduling criteria to manage processes from the same or from different programs, and processes may be moved around among processors as the scheduler dictates. The operating system may extend the uniprocessor scheduling criteria to include multiprocessor-specific issues. In fact, most modern systems fall somewhere between the above two extremes: The user may ask the system to preserve certain properties, giving the user program some control over the mapping, but the operating system is allowed to change the mapping dynamically for effective resource management.

Mapping and associated resource management issues in multiprogrammed systems are active areas of research. However, our goal here is to understand parallel programming in its basic form, so for simplicity we assume that a single parallel program has complete control over the resources of the machine. We also assume that the number of processes equals the number of processors, and neither changes during the execution of the program. By default, the operating system will place one process on every processor in no particular order. Processes are assumed not to migrate from one processor to another during execution. For this reason, we use the terms "process" and "processor" interchangeably in the rest of the chapter.

2.3.2 Parallelizing Computation versus Data

The view of the parallelization process described above has been centered on computation, or work, rather than on data. It is the computation that is decomposed and assigned. However, due to the communication abstraction or performance considerations, we may be responsible for decomposing and assigning data to processes as well. In fact, in many important classes of prob-

lems the decomposition of work and data are so strongly related that they are difficult or even unnecessary to distinguish. Ocean is a good example: Each cross-sectional grid through the ocean is represented as an array, and we can view the parallelization as decomposing the data in each array and assigning parts of it to processes. The process that is assigned a portion of an array will then be responsible for the computation associated with that portion, a so-called *owner computes* arrangement. A similar situation exists in data mining, where we can view the database as being decomposed and assigned; of course, here there is also the question of assigning the itemsets to processes. Several language systems, including the High Performance Fortran standard [KLS+94, HPF93], allow the programmer to specify the decomposition and assignment of data structures; the assignment of computation then follows the assignment of data in an owner computes manner. However, the distinction between computation and data is stronger in many other applications, including the Barnes-Hut and Raytrace case studies as we shall see. Since the *computation-centric* view is more general, we shall retain this view and consider data management to be part of the orchestration step.

2.3.3 Goals of the Parallelization Process

As stated previously, the major goal of using a parallel machine is to improve performance by obtaining speedup over the best uniprocessor execution. Each of the steps in creating a parallel program has a role to play in achieving that overall goal, and each has its own performance goals.

Table 2-1 Steps in the Parallelization Process and Their Goals

Step	Architecture-dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency, but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce non-inherent communication via data locality (see next chapter) Reduce cost of comm/synch as seen by processor Reduce serialization of shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

These are summarized in Table 2-1, and we shall discuss them in more detail in the next chapter.

Creating an effective parallel program requires evaluating cost as well as performance. In addition to the dollar cost of the machine itself, we must consider the resource requirements of the program on the architecture and the effort it takes to develop a satisfactory program. While costs and their impact are often more difficult to quantify than performance, they are very important and we must not lose sight of them; in fact, we may need to compromise performance to reduce them. As algorithm designers, we should favor high-performance solutions that keep the resource requirements of the algorithm small and that don't require inordinate programming effort. As architects, we should try to design high-performance systems that, in addition to being low-cost, reduce programming effort and facilitate resource-efficient algorithms. For example, an architecture that delivers gradually improving performance with increased programming effort may be

preferable to one that is capable of ultimately delivering better performance but only with inordinate programming effort.

Having understood the basic process and its goals, let us apply it to a simple but detailed example and see what the resulting parallel programs look like in the three major modern programming models introduced in Chapter 1: shared address space, message passing, and data parallel. We shall focus here on illustrating programs and programming primitives, not so much on performance.

2.4 Parallelization of an Example Program

The four case studies introduced at the beginning of the chapter all lead to parallel programs that are too complex and too long to serve as useful sample programs. Instead, this section presents a simplified version of a piece or *kernel* of Ocean: its equation solver. It uses the equation solver to dig deeper and illustrate how to implement a parallel program using the three programming models. Except for the data parallel version, which necessarily uses a high-level data parallel language, the parallel programs are not written in an aesthetically pleasing language that relies on software layers to hide the orchestration and communication abstraction from the programmer. Rather, they are written in C or Pascal-like pseudocode augmented with simple extensions for parallelism, thus exposing the basic communication and synchronization primitives that a shared address space or message passing communication abstraction must provide. Standard sequential languages augmented with primitives for parallelism also reflect the state of most real parallel programming today.

2.4.1 A Simple Example: The Equation Solver Kernel

The equation solver kernel solves a simple partial differential equation on a grid, using what is referred to as a finite differencing method. It operates on a regular, two-dimensional grid or array of *(n+2)*-by-*(n+2)* elements, such as a single horizontal cross-section of the ocean basin in Ocean. The border rows and columns of the grid contain boundary values that do not change, while the interior *n*-by-*n* points are updated by the solver starting from their initial values. The computation proceeds over a number of sweeps. In each sweep, it operates on all the elements of the grid, for each element replacing its value with a weighted average of itself and its four nearest neighbor elements (above, below, left and right, see Figure 2-6). The updates are done in-place in the grid, so a point sees the new values of the points above and to the left of it, and the old values of the points below it and to its right. This form of update is called the Gauss-Seidel method. During each sweep the kernel also computes the average difference of an updated element from its previous value. If this average difference over all elements is smaller than a predefined "tolerance" parameter, the solution is said to have converged and the solver exits at the end of the sweep. Otherwise, it performs another sweep and tests for convergence again. The sequential pseudocode is shown in Figure 2-7. Let us now go through the steps to convert this simple equation solver to a parallel program for each programming model. The decomposition and assignment are essentially the same for all three models, so these steps are examined them in a general context. Once we enter the orchestration phase, the discussion will be organized explicitly by programming model.

Expression for updating each interior point:

$$A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$
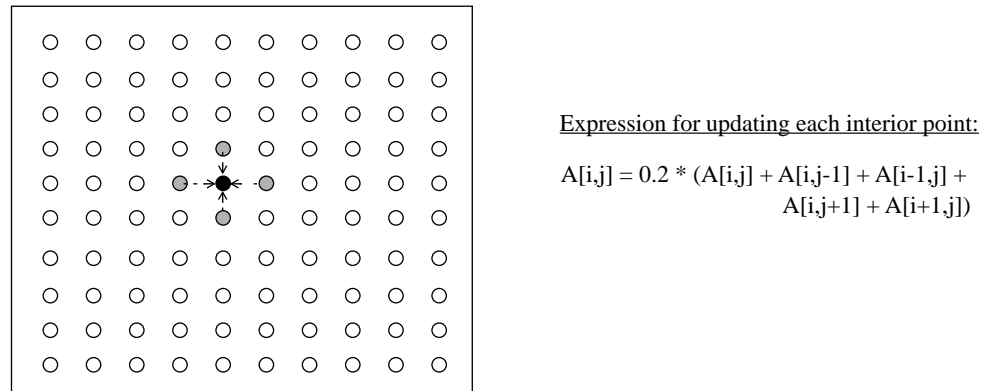
Figure 2-6 Nearest-neighbor update of a grid point in the simple equation solver.

The black point is A[i,j] in the two-dimensional array that represents the grid, and is updated using itself and the four shaded points that are its nearest neighbors according to the equation at the right of the figure.

2.4.2 Decomposition

For programs that are structured in successive loops or loop nests, a simple way to identify concurrency is to start from the loop structure itself. We examine the individual loops or loop nests in the program one at a time, see if their iterations can be performed in parallel, and determine whether this exposes enough concurrency. We can then look for concurrency across loops or take a different approach if necessary. Let us follow this program structure based approach in Figure 2-7.

Each iteration of the outermost loop, beginning at line 15, sweeps through the entire grid. These iterations clearly are not independent, since data that are modified in one iteration are accessed in the next. Consider the loop nest in lines 17-24, and ignore the lines containing `diff`. Look at the inner loop first (the *j* loop starting on line 18). Each iteration of this loop reads the grid point (A[i,j-1]) that was written in the previous iteration. The iterations are therefore sequentially dependent, and we call this a *sequential loop*. The outer loop of this nest is also sequential, since the elements in row *i-1* were written in the previous (*i-1*th) iteration of this loop. So this simple analysis of existing loops and their dependences uncovers no concurrency in this case.

In general, an alternative to relying on program structure to find concurrency is to go back to the fundamental dependences in the underlying algorithms used, regardless of program or loop structure. In the equation solver, we might look at the fundamental data dependences at the granularity of individual grid points. Since the computation proceeds from left to right and top to bottom in the grid, computing a particular grid point in the sequential program uses the updated values of the grid points directly above and to the left. This dependence pattern is shown in Figure 2-8. The result is that the elements along a given anti-diagonal (south-west to north-east) have no dependences among them and can be computed in parallel, while the points in the next anti-diagonal depend on some points in the previous one. From this diagram, we can observe that of the $O(n^2)$

```
1. int n;                                  /* size of matrix: n-by-n elements*/
2. float **A, diff = 0;


3. main()
4. begin
5.   read(n) ;                             /* read input parameter: matrix size*/
6.   A ← malloc (a 2-d array of size n+2 by n+2 doubles);
7.   initialize(A);                        /* initialize the matrix A somehow */
8.   Solve (A);                            /* call the routine to solve equation*/
9. end main


10. procedure Solve (A)                    /* solve the equation system */
11.   float **A;                           /* A is an n+2 by n+2 array*/
12. begin
13.   int i, j, done = 0;
14.   float diff = 0, temp;
15.   while (!done) do                     /* outermost loop over sweeps */
16.       diff = 0;                        /* initialize maximum difference to 0 */
17.       for i ← 1 to n do                /* sweep over non-border points of grid */
18.           for j ← 1 to n do
19.               temp = A[i,j];           /* save old value of element */
20.               A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                       A[i,j+1] + A[i+1,j]);      /*compute average */
22.               diff += abs(A[i,j] - temp);
23.           end for
24.       end for
25.       if (diff/(n*n) < TOL) then done = 1;
26.   end while
27.end procedure
```

Figure 2-7 Pseudocode describing the sequential equation solver kernel.

The main body of work to be done in each iteration is in the nested for loop in lines 17 to 23. This is what we would like to parallelize.

work involved in each sweep, there a sequential dependence proportional to *n* along the diagonal and inherent concurrency proportional to *n*.

Suppose we decide to decompose the work into individual grid points, so updating a single grid point is a task. There are several ways to exploit the concurrency this exposes. Let us examine a few. First, we can leave the loop structure of the program as it is, and insert point-to-point synchronization to ensure that a grid point has been produced in the current sweep before it is used by the points to the right of or below it. Thus, different loop nests and even different sweeps might be in progress simultaneously on different elements, as long as the element-level dependences are not violated. But the overhead of this synchronization at grid-point level may be too
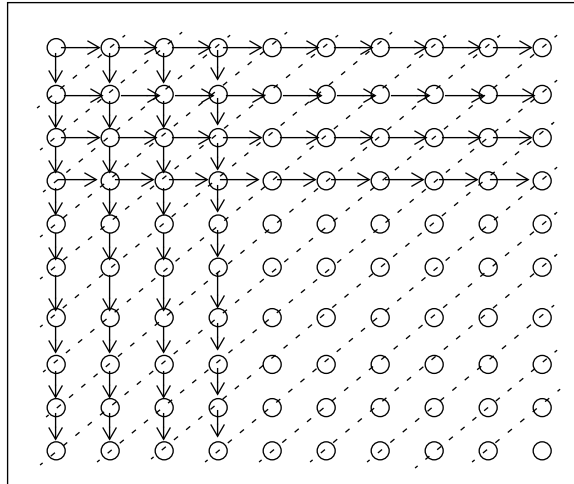
Figure 2-8 Dependences and concurrency in the Gauss-Seidel equation solver computation.

The horizontal and vertical lines with arrows indicate dependences, while the anti-diagonal, dashed lines connect points with no dependences among them and that can be computed in parallel.

high. Second, we can change the loop structure, and have the first for loop (line 17) be over anti-diagonals and the inner for loop be over elements within an anti-diagonal. The inner loop can now be executed completely in parallel, with global synchronization between iterations of the outer for loop to preserve dependences conservatively across anti-diagonals. Communication will be orchestrated very differently in the two cases, particularly if communication is explicit. However, this approach also has problems. Global synchronization is still very frequent: once per anti-diagonal. Also, the number of iterations in the parallel (inner) loop changes with successive outer loop iterations, causing load imbalances among processors especially in the shorter anti-diagonals. Because of the frequency of synchronization, the load imbalances, and the programming complexity, neither of these approaches is used much on modern architectures.

The third and most common approach is based on exploiting knowledge of the problem beyond the sequential program itself. The order in which the grid points are updated in the sequential program (left to right and top to bottom) is not fundamental to the Gauss-Seidel solution method; it is simply one possible ordering that is convenient to program sequentially. Since the Gauss-Seidel method is not an exact solution method (unlike Gaussian elimination) but rather iterates until convergence, we can update the grid points in a different order as long as we use updated values for grid points frequently enough.[1] One such ordering that is used often for parallel versions is called *red-black* ordering. The idea here is to separate the grid points into alternating red points and black points as on a checkerboard (Figure 2-9), so that no red point is adjacent to a black point or vice versa. Since each point reads only its four nearest neighbors, it is clear that to

---

1. Even if we don't use updated values from the current while loop iteration for any grid points, and we always use the values as they were at the end of the previous while loop iteration, the system will still converge, only much slower. This is called Jacobi rather than Gauss-Seidel iteration.
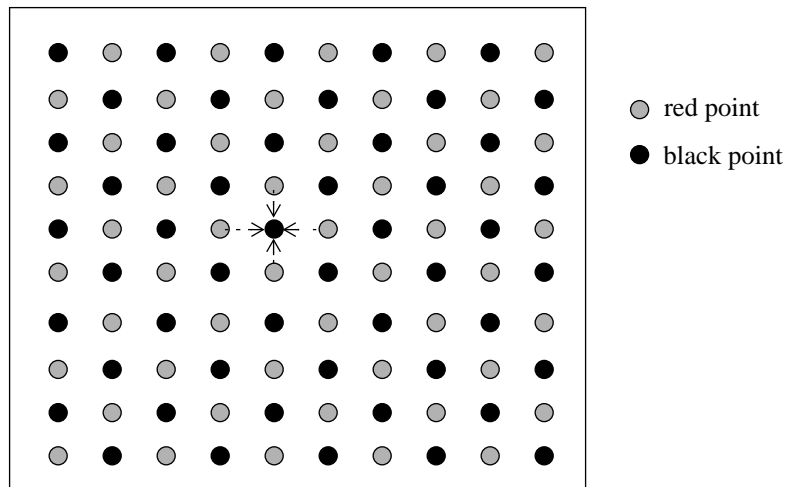
Figure 2-9 Red-black ordering for the equation solver.

The sweep over the grid is broken up into two sub-sweeps: The first computes all the red points, and the second all the black points. SInce red points depend only on black points and vice versa, there are no dependences within a sub-sweep.

compute a red point we do not need the updated value of any other red point, but only the updated values of the above and left black points (in a standard sweep), and vice versa. We can therefore divide a grid sweep into two phases: first computing all red points and then computing all black points. Within each phase there are no dependences among grid points, so we can compute all $\frac{n^2}{2}$ red points in parallel, then synchronize globally, and then compute all $\frac{n^2}{2}$ black points in parallel. Global synchronization is conservative and can be replaced by point-to-point synchronization at the level of grid points—since not all black points need to wait for all red points to be computed—but it is convenient.

The red-black ordering is different from our original sequential ordering, and can therefore both converge in fewer or more sweeps as well as produce different final values for the grid points (though still within the convergence tolerance). Note that the black points will see the updated values of all their (red) neighbors in the current sweep, not just the ones to the left and above. Whether the new order is sequentially better or worse than the old depends on the problem. The red-black order also has the advantage that the values produced and convergence properties are independent of the number of processors used, since there are no dependences within a phase. If the sequential program itself uses a red-black ordering then parallelism does not change the properties at all.

The solver used in Ocean in fact uses a red-black ordering, as we shall see later. However, red-black ordering produces a longer kernel of code than is appropriate for illustration here. Let us therefore examine a simpler but still common asynchronous method that does not separate points into red and black. Global synchronization is used between grid sweeps as above, and the loop structure for a sweep is not changed from the top-to-bottom, left-to-right order. Instead, within a sweep a process simply updates the values of all its assigned grid points, accessing its nearest

neighbors whether or not they have been updated in the current sweep by their assigned processes or not. That is, it ignores dependences among grid points within a sweep. When only a single process is used, this defaults to the original sequential ordering. When multiple processes are used, the ordering is unpredictable; it depends on the assignment of points to processes, the number of processes used, and how quickly different processes execute at runtime. The execution is no longer deterministic, and the number of sweeps required to converge may depend on the number of processors used; however, for most reasonable assignments the number of sweeps will not vary much.

If we choose a decomposition into individual inner loop iterations (grid points), we can express the program by writing lines 15 to 26 of Figure 2-7 as shown in Figure 2-10. All that we have

```
15.    while (!done) do                              /* a sequential loop*/
16.         diff = 0;
17.         for all i ← 1 to n do                    /* a parallel loop nest */
18.             for all j ← 1 to n do
19.                 temp = A[i,j];
20.                 A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);
22.                 diff += abs(A[i,j] - temp);
23.             end for_all
24.         end for_all
25.         if (diff/(n*n) < TOL) then done = 1;
26.    end while
```

Figure 2-10 Parallel equation solver kernel with decomposition into elements and no explicit assignment.

Since both for loops are made parallel by using for_all instead of for, the decomposition is into individual grid elements. Other than this change, the code is the same as the sequential code.

done is replace the keyword for in the parallel loops with for_all. A for_all loop simply tells the underlying software/hardware system that all iterations of the loop can be executed in parallel without dependences, but says nothing about assignment. A loop nest with both nesting levels being for_all means that all iterations in the loop nest ($n*n$ or n^2 here) can be executed in parallel. The system can orchestrate the parallelism in any way it chooses; the program does not take a position on this. All it assumes is that there is an implicit global synchronization after a for_all loop nest.

In fact, we can decompose the computation into not just individual iterations as above but any aggregated groups of iterations we desire. Notice that decomposing the computation corresponds very closely to decomposing the grid itself. Suppose now that we wanted to decompose into rows of elements instead, so that the work for an entire row is an indivisible task which must be assigned to the same process. We could express this by making the inner loop on line 18 a sequential loop, changing its for_all back to a for, but leaving the loop over rows on line 17 as a parallel for_all loop. The parallelism, or *degree of concurrency*, exploited under this decomposition is reduced from n^2 inherent in the problem to n: Instead of n^2 independent tasks of duration one unit each, we now have n independent tasks of duration n units each. If each task

executed on a different processor, we would have approximately $2n$ communication operations (accesses to data computed by other tasks) for n points.

Let us proceed past decomposition and see how we might assign rows to processes explicitly, using a row-based decomposition.

2.4.3 Assignment

The simplest option is a static (predetermined) assignment in which each processor is responsible for a contiguous block of rows, as shown in Figure 2-11. Row i is assigned to process $\left\lfloor \frac{i}{P} \right\rfloor$.

Alternative static assignments to this so-called *block* assignment are also possible, such as a *cyclic* assignment in which rows are interleaved among processes (process i is assigned rows i, $i + P$, and so on). We might also consider a dynamic assignment, where each process repeatedly grabs the next available (not yet computed) row after it finishes with a row task, so it is not predetermined which process computes which rows. For now, we will work with the static block assignment. This simple partitioning of the problem exhibits good load balance across processes as long as the number of rows is divisible by the number of processes, since the work per row is uniform. Observe that the static assignments have further reduced the parallelism or degree of concurrency, from n to p, and the block assignment has reduced the communication required by assigning adjacent rows to the same processor. The communication to computation ratio is now only $O\left(\frac{n}{p}\right)$.
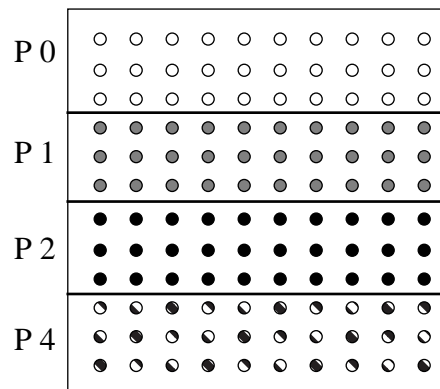
Figure 2-11 A simple assignment for the parallel equation solver.

Each of the four processors is assigned a contiguous, equal number of rows of the grid. In each sweep, it will perform the work needed to update the elements of its assigned rows.

Given this decomposition and assignment, we are ready to dig into the orchestration phase. This requires that we pin down the programming model. We begin with a high level, data parallel model. Then, we examine the two major programming models that this and other models might compile down to: a shared address space and explicit message passing.

2.4.4 Orchestration under the Data Parallel Model

The data parallel model is convenient for the equation solver kernel, since it is natural to view the computation as a single thread of control performing global transformations on a large array data structure, just as we have done above [Hil85,HiS86]. Computation and data are quite interchangeable, a simple decomposition and assignment of the data leads to good load balance across processes, and the appropriate assignments are very regular in shape and can be described by simple primitives. Pseudocode for the data-parallel equation solver is shown in Figure 2-12. We assume that global declarations (outside any procedure) describe shared data, and all other data are private to a process. Dynamically allocated shared data, such as the array A, are allocated with a G_MALLOC (global malloc) call rather than a regular malloc. Other than this, the main differences from the sequential program are the use of an DECOMP statement, the use of for_all loops instead of for loops, the use of a private mydiff variable per process, and the use of a REDUCE statement. We have already seen that for_all loops specify that the iterations can be performed in parallel. The DECOMP statement has a two-fold purpose. First, it specifies the assignment of the iterations to processes (DECOMP is in this sense an unfortunate choice of word). Here, it is a [BLOCK, *, nprocs] assignment, which means that the first dimension (rows) is partitioned into contiguous pieces among the nprocs processes, and the second dimension is not partitioned at all. Specifying [CYCLIC, *, nprocs] would have implied a cyclic or interleaved partitioning of rows among nprocs processes, specifying [BLOCK, BLOCK, nprocs] would have implied a subblock decomposition, and specifying [*, CYCLIC, nprocs] would have implied an interleaved partitioning of columns. The second and related purpose of DECOMP is that it also specifies how the grid data should be distributed among memories on a distributed-memory machine (this is restricted to be the same as the assignment in most current data-parallel languages, following the owner computes rule, which works well in this example).

The mydiff variable is used to allow each process to first independently compute the sum of the difference values for its assigned grid points. Then, the REDUCE statement directs the system to add all their partial mydiff values together into the shared diff variable. The reduction operation may be implemented in a library in a manner best suited to the underlying architecture.

While the data parallel programming model is well-suited to specifying partitioning and data distribution for regular computations on large arrays of data, such as the equation solver kernel or the Ocean application, the desirable properties do not hold true for more irregular applications, particularly those in which the distribution of work among tasks or the communication pattern changes unpredictably with time. For example, think of the stars in Barnes-Hut or the rays in Raytrace. Let us look at the more flexible, lower-level programming models in which processes have their own individual threads of control and communicate with each other when they please.

2.4.5 Orchestration under the Shared Address Space Model

In a shared address space we can simply declare the matrix *A* as a single shared array—as we did in the data parallel model—and processes can reference the parts of it they need using loads and stores with exactly the same array indices as in a sequential program. Communication will be generated implicitly as necessary. With explicit parallel processes, we now need mechanisms to create the processes, coordinate them through synchronization, and control the assignment of

```
1.  int n, nprocs;                    /* grid size (n+2-by-n+2) and number of processes*/
2.  float **A, diff = 0;


3.  main()
4.  begin
5.    read(n); read(nprocs);;         /* read input grid size and number of processes*/
6.    A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.    initialize(A);                   /* initialize the matrix A somehow */
8.    Solve (A);                       /* call the routine to solve equation*/
9.  end main


10. procedure Solve(A)                 /* solve the equation system */
11.   float **A;                       /* A is an n+2 by n+2 array*/
12. begin
13.   int i, j, done = 0;
14.   float mydiff = 0, temp;
14a.  DECOMP A[BLOCK,*];
15.   while (!done) do                 /* outermost loop over sweeps */
16.       mydiff = 0;                  /* initialize maximum difference to 0 */
17.       for_all i ← 1 to n do        /* sweep over non-border points of grid */
18.           for_all j ← 1 to n do
19.               temp = A[i,j];       /* save old value of element */
20.               A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                     A[i,j+1] + A[i+1,j]);      /*compute average */
22.               mydiff += abs(A[i,j] - temp);
23.           end for_all
24.       end for_all
24a.      REDUCE (mydiff, diff, ADD);
25.       if (diff/(n*n) < TOL) then done = 1;
26.   end while
27. end procedure
```

Figure 2-12 Pseudocode describing the data-parallel equation solver.

Differences from the sequential code are shown in italicized bold font. The decomposition is still into individual elements, as indicated by the nested for_all loop. The assignment, indicated by the unfortunately labelled DECOMP statement, is into blocks of contiguous rows (the first or column dimension is partitioned into blocks, and the second or row dimension is not partitioned). The REDUCE statement sums the locally computed mydiffs into a global diff value. The while loop is still serial.

work to processes. The primitives we use are typical of low-level programming environments such as *parmacs* [LO+87], and are summarized in Table 2-2.

Table 2-2 Key Shared Address Space Primitives

Name	Syntax	Function
CREATE	`CREATE(p,proc,args)`	Create `p` processes that start executing at procedure `proc` with arguments `args`.
G_MALLOC	`G_MALLOC(size)`	Allocate shared data of `size` bytes
LOCK	`LOCK(name)`	Acquire mutually exclusive access
UNLOCK	`UNLOCK(name)`	Release mutually exclusive access
BARRIER	`BARRIER(name, number)`	Global synchronization among `number` processes: None gets past BARRIER until `number` have arrived
WAIT_FOR_END	`WAIT_FOR_END(number)`	Wait for `number` processes to terminate
wait for flag	`while (!flag);` or `WAIT(flag)`	Wait for `flag` to be set (spin or block); for point-to-point event synchronization.
set flag	`flag = 1;` or `SIGNAL(flag)`	Set `flag`; wakes up process spinning or blocked on `flag`, if any

Pseudocode for the parallel equation solver in a shared address space is shown in Figure 2-13. The special primitives for parallelism are shown in italicized bold font. They are typically implemented as library calls or macros, each of which expands to a number of instructions that accomplish its goal. Although the code for the Solve procedure is remarkably similar to the sequential version, let's go through it one step at a time.

A single process is first started up by the operating system to execute the program, starting from the procedure called main. Let's call it the *main* process. It reads the input, which specifies the size of the grid *A* (recall that input n denotes an (n+2) -by-(n+2) grid of which n-by-n points are updated by the solver). It then allocates the grid *A* as a two-dimensional array in the shared address space using the G_MALLOC call, and initializes the grid. The G_MALLOC call is similar to a usual `malloc` call—used in the C programming language to allocate data dynamically in the program's heap storage—and it returns a pointer to the allocated data. However, it allocates the data in a shared region of the heap, so they can be accessed and modified by any process. For data that are not dynamically allocated on the heap, different systems make different assumptions about what is shared and what is private to a process. Let us assume that all "global" data in the sequential C programming sense—i.e. data declared outside any procedure, such as `nprocs` and n in Figure 2-13—are shared. Data on a procedure's stack (such as `mymin`, `mymax`, `mydiff`, `temp`, `i` and `j`) are private to a process that executes the procedure, as are data allocated with a regular `malloc` call (and data that are explicitly declared to be private, not used in this program).

Having allocated data and initialized the grid, the program is ready to start solving the system. It therefore creates (`nprocs-1`) "worker" processes, which begin executing at the procedure called Solve. The main process then also calls the Solve procedure, so that all `nprocs` processes

```
1. int n, nprocs;              /* matrix dimension and number of processors to be used */
2a. float **A, diff;           /*A is global (shared) array representing the grid */
                               /* diff is global (shared) maximum difference in current sweep */
2b. LOCKDEC(diff_lock);        /* declaration of lock to enforce mutual exclusion */
2c. BARDEC (bar1);             /* barrier declaration for global synchronization between sweeps*/

3. main()
4. begin
5.    read(n);    read(nprocs);     /* read input matrix size and number of processes*/
6.    A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.    initialize(A);                            /* initialize A in an unspecified way*/
8a.   CREATE (nprocs-1, Solve, A);
8     Solve(A);                      /* main process becomes a worker too*/
8b.   WAIT_FOR_END;                  /* wait for all child processes created to terminate */
9. end main

10. procedure Solve(A)
11. float **A;         /* A is entire n+2-by-n+2 shared array, as in the sequential program */
12. begin
13.   int i,j, pid, done = 0;
14.   float temp, mydiff = 0;               /* private variables/
14a.  int mymin ← 1 + (pid * n/nprocs);     /* assume that n is exactly divisible by */
14b.  int mymax ← mymin + n/nprocs - 1;     /* nprocs for simplicity here*/

15. while (!done) do                         /* outer loop over all diagonal elements */
16.   mydiff = diff = 0;                     /* set global diff to 0 (okay for all to do it) */
17.   for i ← mymin to mymax do              /* for each of my rows */
18.         for j ← 1 to n do                /* for all elements in that row */
19.               temp = A[i,j];
20.               A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                     A[i,j+1] + A[i+1,j]);
22.           mydiff += abs(A[i,j] - temp);
23.         endfor
24.   endfor
25a.  LOCK(diff_lock);                /* update global diff if necessary */
25b.        diff += mydiff;
25c.  UNLOCK(diff_lock);
25d.  BARRIER(bar1, nprocs);                 /* ensure all have got here before checking if done*/

25e. if (diff/(n*n) < TOL) then done = 1; /* check convergence; all get same answer*/
25f. BARRIER(bar1, nprocs);        /* see Exercise c */
26. endwhile
27.end procedure
```

Figure 2-13 Pseudocode describing the parallel equation solver in a shared address space.

Line numbers followed by a letter denote lines that were not present in the sequential version. The numbers are chosen to match the line or control structure in the sequential code with which the new lines are most closely related. The design of the data structures does not have to change from the sequential program. Processes are created with the CREATE call, and the main process waits for them to terminate at the end of the program with the WAIT_FOR_END call. The decomposition is into rows, since the inner loop is unmodified, and the outer loop specifies the assignment of rows to processes. Barriers are used to separate sweeps (and to separate the convergence test from further modification of the global diff variable), and locks are used to provide mutually exclusive access to the global diff variable.

enter the procedure in parallel as equal partners. All created processes execute the same code image until they exit from the program and terminate. This does not mean that they proceed in lock-step or even execute the same instructions, since in general they may follow different control paths through the code. That is, we use a structured, *single-program-multiple-data* style of programming. Control over the assignment of work to processes—and which data they access—is maintained by a few private variables that acquire different values for different processes (e.g. `mymin` and `mymax`), and by simple manipulations of loop control variables. For example, we assume that every process upon creation obtains a unique process identifier (`pid`) between 0 and `nprocs`-1 in its private address space, and in lines 12-13 uses this pid to determine which rows are assigned to it. Processes synchronize through calls to synchronization primitives. We shall discuss these primitives shortly

We assume for simplicity that the total number of rows `n` is an integer multiple of the number of processes `nprocs`, so that every process is assigned the same number of rows. Each process calculates the indices of the first and last rows of its assigned block in the private variables `mymin` and `mymax`. It then proceeds to the actual solution loop.

The outermost while loop (line 21) is still over successive grid sweeps. Although the iterations of this loop proceed sequentially, each iteration or sweep is itself executed in parallel by all processes. The decision of whether to execute the next iteration of this loop is taken separately by each process (by setting the `done` variable and computing the `while (!done)` condition) even though each will make the same decision: The redundant work performed here is very small compared to the cost of communicating a completion flag or the diff value.

The code that performs the actual updates (lines 19-22) is essentially identical to that in the sequential program. Other than the bounds in the loop control statements, used for assignment, the only difference is that each process maintains its own private variable `mydiff`. This private variable keeps track of the total difference between new and old values for only its assigned grid points. It is accumulated it once into the shared `diff` variable at the end of the sweep, rather than adding directly into the shared variable for every grid point. In addition to the serialization and concurrency reason discussed in Section 2.3.1, all processes repeatedly modifying and reading the same shared variable causes a lot of expensive communication, so we do not want to do this once per grid point.

The interesting aspect of the rest of the program (line 23 onward) is synchronization, both mutual exclusion and event synchronization, so the rest of the discussion will focus on it. First, the accumulations into the shared variable by different processes have to be mutually exclusive. To see why, consider the sequence of instructions that a processor executes to add its `mydiff` variable (maintained say in register `r2`) into the shared `diff` variable; i.e. to execute the source statement `diff += mydiff`:

```
load the value of diff into register r1
add the register r2 to register r1
store the value of register r1 into diff
```

Suppose the value in the variable `diff` is 0 to begin with, and the value of `mydiff` in each process is 1. After two processes have executed this code we would expect the value in `diff` to be

2. However, it may turn out to be 1 instead if the processes happen to execute their operations in the interleaved order shown below.

P1	P2	
r1 ← diff		*{P1 gets 0 in its r1}*
	r1 ← diff	*{P2 also gets 0}*
r1 ← r1+r2		*{P1 sets its r1 to 1}*
	r1 ← r1+r2	*{P2 sets its r1 to 1}*
diff ← r1		*{P1 sets cell_cost to 1}*
	diff ← r1	*{P2 also sets cell_cost to 1}*

This is not what we intended. The problem is that a process (here P2) may be able to read the value of the logically shared `diff` between the time that another process (P1) reads it and writes it back. To prohibit this interleaving of operations, we would like the sets of operations from different processes to execute *atomically* (mutually exclusively) with respect to one another. The set of operations we want to execute atomically or mutually exclusively is called a *critical section*.: Once a process starts to execute the first of its three instructions above (its critical section), no other process can execute any of the instructions in its corresponding critical section until the former process has completed the last instruction in the critical section. The LOCK-UNLOCK pair around line 25b achieves mutual exclusion.

A lock, such as `cell_lock`, can be viewed as a shared token that confers an exclusive right. Acquiring the lock through the LOCK primitive gives a process the right to execute the critical section. The process that holds the lock frees it when it has completed the critical section by issuing an UNLOCK command. At this point, the lock is either free for another process to acquire or be granted, depending on the implementation. The LOCK and UNLOCK primitives must be implemented in a way that guarantees mutual exclusion. Our LOCK primitive takes as argument the name of the lock being used. Associating names with locks allows us to use different locks to protect unrelated critical sections, reducing contention and serialization.

Locks are expensive, and even a given lock can cause contention and serialization if multiple processes try to access it at the same time. This is another reason to use a private `mydiff` variable to reduce accesses to the shared variable. This technique is used in many operations called *reductions* (implemented by the REDUCE operation in the data parallel program). A reduction is a situation in which many processes (all, in a global reduction) perform associative operations (such as addition, taking the maximum, etc.) on the same logically shared data. Associativity implies that the order of the operations does not matter. Floating point operations, such as the ones here, are strictly speaking not associative, since how rounding errors accumulate depends on the order of operations. However, the effects are small and we usually ignore them, especially in iterative calculations that are anyway approximations.

Once a process has added its `mydiff` into the global `diff`, it waits until all processes have done so and the value contained in `diff` is indeed the total difference over all grid points. This requires global event synchronization implemented here with a BARRIER. A barrier operation

takes as an argument the name of the barrier and the number of processes involved in the synchronization, and is issued by all those processes. Its semantics are as follows. When a process calls the barrier, it registers the fact that it has reached that point in the program. The process is not allowed to proceed past the barrier call until the specified number of processes participating in the barrier have issued the barrier operation. That is, the semantics of BARRIER(name,p) are: wait until p processes get here and only then proceed.

Barriers are often used to separate distinct phases of computation in a program. For example, in the Barnes-Hut galaxy simulation we use a barrier between updating the positions of the stars at the end of one time-step and using them to compute forces at the beginning of the next one, and in data mining we may use a barrier between counting occurrences of candidate itemsets and using the resulting large itemsets to generate the next list of candidates. Since they implement all-to-all event synchronization, barriers are usually a conservative way of preserving dependences; usually, not all operations (or processes) after the barrier actually need to wait for all operations before the barrier. More specific, point-to-point or group event synchronization would enable some processes to get past their synchronization event earlier; however, from a programming viewpoint it is often more convenient to use a single barrier than to orchestrate the actual dependences through point-to-point synchronization.

When point-to-point synchronization is needed, one way to orchestrate it in a shared address space is with wait and signal operations on semaphores, with which we are familiar from operating systems. A more common way is by using normal shared variables as synchronization *flags*, as shown in Figure 2-14. Since *P1* simply spins around in a tight while loop for something to

P1	P2
	A = 1;
a: while (flag is 0) do nothing;	b: flag = 1;
print A;	

Figure 2-14 Point-to-point event synchronization using flags.

Suppose we want to ensure that a process P1 does not get past a certain point (say a) in the program until some other process P2 has already reached another point (say b). Assume that the variable flag (and A) was initialized to 0 before the processes arrive at this scenario. If P1 gets to statement a after P2 has already executed statement b, P1 will simply pass point a. If, on the other hand, P2 has not yet executed b, then P1 will remain in the "idle" while loop until P2 reaches b and sets flag to 1, at which point P1 will exit the idle loop and proceed. If we assume that the writes by P2 are seen by P1 in the order in which they are performed, then this synchronization will ensure that P1 prints the value 1 for A.

happen, keeping the processor busy during this time, we call this *spin-waiting* or *busy-waiting*. Recall that in the case of a semaphore the waiting process does not spin and consume processor resources but rather blocks (suspends) itself, and is awoken when the other process signals the semaphore.

In event synchronization among subsets of processes, or *group event synchronization*, one or more processes may act as producers and one or more as consumers. Group event synchronization can be orchestrated either using ordinary shared variables as flags or by using barriers among subsets of processes.

Once it is past the barrier, a process reads the value of diff and examines whether the average difference over all grid points (diff/(n*n)) is less than the error tolerance used to determine convergence. If so, it sets the done flag to exit from the while loop; if not, it goes on to perform another sweep.

Finally, the WAIT_FOR_END called by the main process at the end of the program (line 11) is a particular form of all-to-one synchronization. Through it, the main process waits for all the worker processes it created to terminate. The other processes do not call WAIT_FOR_END, but implicitly participate in the synchronization by terminating when they exit the Solve procedure that was their entry point into the program.

In summary, for this simple equation solver the parallel program in a shared address space is not too different in structure from the sequential program. The major differences are in the control flow—which are implemented by changing the bounds on some loops—in creating processes and in the use of simple and generic synchronization primitives. The body of the computational loop is unchanged, as are the major data structures and references to them. Given a strategy for decomposition, assignment and synchronization, inserting the necessary primitives and making the necessary modifications is quite mechanical in this example. Changes to decomposition and assignment are also easy to incorporate. While many simple programs have this property in a shared address space we will see later that more substantial changes are needed as we seek to obtain higher and higher parallel performance, and also as we address more complex parallel programs.

Example 2-2   How would the code for the shared address space parallel version of the equation solver change if we retained the same decomposition into rows but changed to a cyclic or interleaved assignment of rows to processes?

Answer   Figure 2-15 shows the relevant pseudocode. All we have changed in the code is the

```
17.    for i ← pid+1 to n by nprocs do    /* for my interleaved set of rows*/
18.        for j ← 1 to n do                /* for all elements in that row */
19.            temp = A[i,j];
20.            A[i,j] ¨ 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                A[i,j+1] + A[i+1,j]);
22.            mydiff += abs(A[i,j] - temp);
23.        endfor
24.    endfor
```

Figure 2-15 Cyclic assignment of row-based solver in a shared address space.

All that changes in the code from the block assignment of rows in Figure 2-13 is first for statement in line 17. The data structures or accesses to them do not have to be changed.

control arithmetic in line 17. The same global data structure is used with the same indexing, and all the rest of the parallel program stays exactly the same.

2.4.6 Orchestration under the Message Passing Model

We now examine a possible implementation of the parallel solver using explicit message passing between private address spaces, employing the same decomposition and assignment as before. Since we no longer have a shared address space, we cannot simply declare the matrix *A* to be shared and have processes reference parts of it as they would in a sequential program. Rather, the logical data structure *A* must be represented by smaller data structures, which are allocated among the private address spaces of the cooperating processes in accordance with the assignment of work. The process that is assigned a block of rows allocates those rows as a sub-grid in its local address space.

The message-passing program shown in Figure 2-16 is structurally very similar to the shared address space program in Figure 2-13 (more complex programs will reveal further differences in the next chapter, see Section 3.7). Here too, a main process is started by the operating system when the program executable is invoked, and this main process creates (nprocs-1) other processes to collaborate with it. We assume that every created process acquires a process identifier pid between 0 and nprocs-1, and that the CREATE call allows us to communicate the program's input parameters (n and nprocs) to the address space of each process.[1] The outermost

---

1. An alternative organization is to use what is called a "hostless" model, in which there is no single main process. The number of processes to be used is specified to the system when the program is invoked. The system then starts up that many processes and distributes the code to the relevant processing nodes. There is no need for a CREATE primitive in the program itself, every process reads the program inputs (n and nprocs) separately, though processes still acquire unique user-level pid's.

```
1. int pid, n, nprocs;          /* process id, matrix dimension and number of processors to be used */
2. float **myA;
3. main()
4. begin
5.    read(n);  read(nprocs);            /* read input matrix size and number of processes*/
8a.   CREATE (nprocs-1 processes that start at procedure Solve);
8b.   Solve();                           /* main process becomes a worker too*/
8c.   WAIT_FOR_END;                      /* wait for all child processes created to terminate */
9. end main

10. procedure Solve()
11. begin
13.   int i,j, pid, n' = n/nprocs, done = 0;
14.   float temp, tempdiff, mydiff = 0;       /* private variables/
6.    myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);/* my assigned rows of A */
7.    initialize(myA);                        /* initialize my rows of A, in an unspecified way*/


15.while (!done) do
16.   mydiff = 0;                               /* set local diff to 0 */
16a.if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.  if (pid = nprocs-1) then SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.  if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.  if (pid != nprocs-1) then RECEIVE(&myA[n'+1,0],n*sizeof(float),pid+1,ROW);
              /* border rows of neighbors have now been copied into myA[0,*] and myA[n'+1,*] */
17.   for i ← 1 to n' do                       /* for each of my rows */
18.       for j ← 1 to n do                     /* for all elements in that row */
19.           temp = myA[i,j];
20.           myA[i,j] ¨ 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.               myA[i,j+1] + myA[i+1,j]);
22.           mydiff += abs(myA[i,j] - temp);
23.       endfor
24.   endfor
      /* communicate local diff values and obtain determine if done; can be replaced by reduction and broadcast */
25a.  if (pid != 0) then                        /* process 0 holds global total diff*/
25b.      SEND(mydiff,sizeof(float),0,DIFF);
25c.      RECEIVE(mydiff,sizeof(float),0,DONE);
25d.  else
25e.      for i ← 1 to nprocs-1 do              /* for each of my rows */
25f.          RECEIVE(tempdiff,sizeof(float),*,DONE);
25g.          mydiff += tempdiff;               /* accumulate into total */
25h.      endfor
25i.      for i ← 1 to nprocs-1 do              /* for each of my rows */
25j.          SEND(done,sizeof(int),i,DONE);
25k.      endfor
25l.  endif
26.       if (mydiff/(n*n) < TOL) then done = 1;
27. endwhile
28. end procedure
```

Figure 2-16 Pseudocode describing parallel equation solver with explicit message-passing.

Now the meaning of the data structures and the indexing of them changes in going to the parallel code. Each process has its own myA data structure that represents its assigned part of the grid, and myA[i,j] referenced by different processes refers to different parts of the logical overall grid. The communication is all contained in lines 16 a-d and 25 a-f. No locks or barriers are needed, since the synchronization is implicit in the send-receive pairs. Several extra lines of code are added to orchestrate the communication with simple sends and receives.

loop of the `Solve` routine (line 15) still iterates over grid sweeps until convergence. In every iteration a process performs the computation for its assigned rows and communicates as necessary. The major differences are in the data structures used to represent the logically shared matrix *A* and in how interprocess communication is implemented. We shall focus on these differences.

Instead of representing the matrix to be factored as a single global (n+2)-by-(n+2) array *A*, each process in the message-passing program allocates an array called `myA` of size (`nprocs/n +` `2`)-by-(`nprocs/n  +  2`) in its private address space. This array represents its assigned `nprocs/n` rows of the logically shared matrix *A*, plus two rows at the edges to hold the boundary data from its neighboring partitions. The boundary rows from its neighbors must be communicated to it explicitly and copied into these extra or *ghost* rows. Ghost rows are used because without them the communicated data would have to received into separate one-dimensional arrays with different names created specially for this purpose, which would complicate the referencing of the data when they are read in the inner loop (lines 20-21). Communicated data have to be copied into the receiver's private address space anyway, so programming is made easier by extending the existing data structure rather than allocating new ones.

Recall from Chapter 1 that both communication and synchronization in a message passing program are based on two primitives: `SEND` and `RECEIVE`. The program event that initiates data transfer is the `SEND` operation, unlike in a shared address space where data transfer is usually initiated by the consumer or receiver using a load instruction. When a message arrives at the destination processor, it is either kept in the network queue or temporarily stored in a system buffer until a process running on the destination processor posts a `RECEIVE` for it. With a `RECEIVE`, a process reads an incoming message from the network or system buffer into a designated portion of the private (application) address space. A `RECEIVE` does not in itself cause any data to be transferred across the network.

Our simple SEND and RECEIVE primitives assume that the data being transferred are in a contiguous region of the virtual address space. The arguments in our simple `SEND` call are the start address—in the sending processor's private address space—of the data to be sent, the size of the message in bytes, the pid of the destination process—which we must be able to name now, unlike in a shared address space—and an optional tag or type associated with the message for matching at the receiver. The arguments to the `RECEIVE` call are a local address at which to place the received data, the size of the message, the sender's process id, and the optional message tag or type. The sender's id and the tag, if present, are used to perform a match with the messages that have arrived and are in the system buffer, to see which one corresponds to the receive. Either or both of these fields may be wild-cards, in which case they will match a message from any source process or with any tag, respectively. SEND and RECEIVE primitives are usually implemented in a library on a specific architecture, just like `BARRIER` and `LOCK` in a shared address space. A full set of message passing primitives commonly used in real programs is part of a standard called the Message Passing Interface (MPI), described at different levels of detail in [Pac96, MPI93, GLS94]. A significant extension is transfers of non-contiguous regions of memory, either with regular stride—such as every tenth word in between addresses *a* and *b*, or four words every sixth word—or by using index arrays to specify unstructured addresses from which to *gather* data on the sending side or to which to *scatter* data on the receiving side. Another is a large degree of flexibility in specifying tags to match messages and in the potential complexity of a match. For example, processes may be divided into groups that communicate certain types of messages only to each other, and collective communication operations may be provided as described below.

Semantically, the simplest forms of SEND and RECEIVE we can use in our program are the so called *synchronous* forms. A synchronous SEND returns control to the calling process only when it is clear that the corresponding RECEIVE has been performed. A synchronous RECEIVE returns control when the data have been received into the destination buffer. Using synchronous messages, our implementation of the communication in lines 16a-d is actually deadlocked. All processors do their SEND first and stall until the corresponding receive is performed, so none will ever get to actually perform their RECEIVE! In general, synchronous message passing can easily deadlock on pairwise exchanges of data if we are not careful. One way to avoid this problem is to have every alternate processor do its SENDs first followed by its RECEIVEs, and the others start do their RECEIVEs first followed by their SENDs. The alternative is to use different semantic flavors of send and receive, as we will see shortly.

The communication is done all at once at the beginning of each iteration, rather than element-by-element as needed in a shared address space. It could be done element by element, but the overhead of send and receive operations is usually too large to make this approach perform reasonably. As a result, unlike in the shared address space version there is no computational asynchrony in the message-passing program: Even though one process updates its boundary rows while its neighbor is computing in the same sweep, the neighbor is guaranteed not see the updates in the current sweep since they are not in its address space. A process therefore sees the values in its neighbors' boundary rows as they were at the end of the previous sweep, which may cause more sweeps to be needed for convergence as per our earlier discussion (red-black ordering would have been particularly useful here).

Once a process has received its neighbors' boundary rows into its ghost rows, it can update its assigned points using code almost exactly like that in the sequential and shared address space programs. Although we use a different name (myA) for a process's local array than the A used in the sequential and shared address space programs, this is just to distinguish it from the logically shared entire grid *A* which is here only conceptual; we could just as well have used the name A). The loop bounds are different, extending from 1 to nprocs/n (substituted by n' in the code) rather than 0 to n-1 as in the sequential program or mymin to mymax in the shared address space program. In fact, the indices used to reference myA are local indices, which are different than the global indices that would be used if the entire logically shared grid A could be referenced as a single shared array. That is, the myA[1,j] reference by different processes refers to different rows of the logically shared grid A. The use of local index spaces can be somewhat more tricky in cases where a global index must also be used explicitly, as seen in Exercise 2.4.

Synchronization, including the accumulation of private mydiff variables into a logically shared diff variable and the evaluation of the done condition, is performed very differently here than in a shared address space. Given our simple synchronous sends and receives which block the issuing process until they complete, the send-receive match encapsulates a synchronization event, and no special operations (like locks and barriers) or additional variables are needed to orchestrate mutual exclusion or event synchronization. Consider mutual exclusion. The logically shared diff variable must be allocated in some process's private address space (here process 0). The identity of this process must be known to all the others. Every process sends its mydiff value to process 0, which receives them all and adds them to the logically shared global diff. Since only it can manipulate this logically shared variable mutual exclusion and serialization are natural and no locks are needed. In fact, process 0 can simply use its own mydiff variable as the global diff.

Now consider the global event synchronization for determining the done condition. Once it has received the `mydiff` values from all the other processes and accumulated them, it sends the accumulated value to all the other processes, which are waiting for it with receive calls. There is no need for a barrier since the completion of the receive implies that all processes's `mydiffs` have been accumulated since process 0 has sent out the result. The processes then compute the done condition to determine whether or not to proceed with another sweep. We could instead have had process 0 alone compute the done condition and then send the `done` variable, not the `diff` value, to the others, saving the redundant calculation of the done condition. We could, of course, implement lock and barrier calls using messages if that is more convenient for programming, although that may lead to request-reply communication and more round-trip messages. More complex send-receive semantics than the synchronous ones we have used here may require additional synchronization beyond the messages themselves, as we shall see.

Notice that the code for the accumulation and done condition evaluation communication has expanded to several lines when using only point-to-point SENDs and RECEIVEs as communication operations. In practice, programming environments would provide library functions like REDUCE (accumulate values from private variables in multiple processes to a single variable in a given process) and BROADCAST (send from one process to all processes) to the programmer which the application processes could use directly to simplify the code in these stylized situations. Using these, lines 25a through 25b in Figure 2-16 can be replaced by the two lines in Figure 2-17. The system may provide special support to improve the performance of these and other *collective communication* operations (such as *multicast* from one to several or even several to several processes, or all-to-all communication in which every process transfers data to every other process), for example by reducing the software overhead at the sender to that of a single message, or they may be built on top of the usual point to point send and receive in user-level libraries for programming convenience only.

```
      /* communicate local diff values and determine if done, using reduction and broadcast */
25b.  REDUCE(0,mydiff,sizeof(float),ADD);
25c.  if (pid == 0) then
25i.        if (mydiff/(n*n) < TOL) then done = 1;
26.   endif
25k.  BROADCAST(0,done,sizeof(int),DONE);
```

Figure 2-17 Accumulation and convergence determination in the solver using REDUCE and BROADCAST instead of SEND and RECEIVE.

The first argument to the REDUCE call is the destination process. All but this process will do a send to this process in the implementation of REDUCE, while this process will do a receive. The next argument is the private variable to be reduced from (in all other processes than the destination) and to (in the destination process), and the third argument is the size of this variable. The last argument is the function to be performed on the variables in the reduction. Similarly, the first argument of the BROADCAST call is the sender; this process does a send and all others do a receive. The second argument is the variable to be broadcast and received into, and the third is its size. The final argument is the optional message type.

Finally, we said earlier that send and receive operations come in different semantic flavors, which we could use to solve our deadlock problem. Let us examine this a little further. The main axis along which these flavors differ is their completion semantics; i.e. when they return control to the user process that issued the send or receive. These semantics affect when the data structures or buffers they use can be reused without compromising correctness. There are two major kinds of send/receive—*synchronous* and *asynchronous*. Within the class of asynchronous mes-

sages, there are two major kinds: *blocking* and *non-blocking*. Let us examine these and see how they might be used in our program.

Synchronous sends and receives are what we assumed above, since they have the simplest semantics for a programmer. A synchronous send returns control to the calling process only when the corresponding synchronous receive at the destination end has completed successfully and returned an acknowledgment to the sender. Until the acknowledgment is received, the sending process is suspended and cannot execute any code that follows the send. Receipt of the acknowledgment implies that the receiver has retrieved the entire message from the system buffer into application space. Thus, the completion of the send guarantees (barring hardware errors) that the message has been successfully received, and that all associated data structures and buffers can be reused.

A *blocking asynchronous* send returns control to the calling process when the message has been taken from the sending application's source data structure and is therefore in the care of the system. This means that when control is returned, the sending process can modify the source data structure without affecting that message. Compared to a synchronous send, this allows the sending process to resume sooner, but the return of control does not guarantee that the message will actually be delivered to the appropriate process. Obtaining such a guarantee would require additional handshaking. A blocking asynchronous receive is similar in that it returns control to the calling process only when the data it is receiving have been successfully removed from the system buffer and placed at the designated application address. Once it returns, the application can immediately use the data in the specified application buffer. Unlike a synchronous receive, a blocking receive does not send an acknowledgment to the sender.

The *nonblocking* asynchronous send and receive allow the greatest overlap between computation and message passing by returning control most quickly to the calling process. A nonblocking send returns control immediately. A nonblocking receive returns control after simply posting the intent to receive; the actual receipt of the message and placement into a specified application data structure or buffer is performed asynchronously at an undetermined time by the system, on the basis of the posted receive. In both the nonblocking send and receive, however, the return of control does not imply anything about the state of the message or the application data structures it uses, so it is the user's responsibility to determine that state when necessary. The state can be determined through separate calls to primitives that probe (query) the state. Nonblocking messages are thus typically used in a two-phase manner: first the send/receive operation itself, and then the probes. The probes—which must be provided by the message-passing library—might either block until the desired state is observed, or might return control immediately and simply report what state was observed.

Which kind of send/receive semantics we choose depends on how the program uses its data structures, and on how much we want to optimize performance over ease of programming and portability to systems with other semantics. The semantics mostly affects event synchronization, since mutual exclusion falls out naturally from having only private address spaces. In the equation solver example, using asynchronous sends would avoid the deadlock problem since processes would proceed past the send and to the receive. However, if we used nonblocking asynchronous receives, we would have to use a probe before actually using the data specified in the receive.

Note that a blocking send/receive is equivalent to a nonblocking send/receive followed immediately by a blocking probe.

Table 2-3 Some Basic Message Passing Primitives.

Name	Syntax	Function
CREATE	CREATE(procedure)	Create process that starts at procedure
BARRIER	BARRIER(name, number)	Global synchronization among number processes: None gets past BARRIER until number have arrived
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate
SEND	SEND(src_addr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier
RECEIVE	RECEIVE(buffer_addr, size, src, tag)	Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr
SEND_PROBE	SEND_PROBE(tag, dest)	Check if message with identifier tag has been sent to process dest (only for asynchronous message passing, and meaning depends on semantics discussed above)
RECV_PROBE	RECV_PROBE(tag, src)	Check if message with identifier tag has been received from process src (only for asynchronous message passing, and meaning depends on semantics)

We leave it as an exercise to transform the message passing version to use a cyclic assignment, as was done for the shared address space version in Example 2-2. The point to observe in that case is that while the two message-passing versions will look syntactically similar, the meaning of the myA data structure will be completely different. In one case it is a section of the global array and in the other is a set of widely separated rows. Only by careful inspection of the data structures and communication patterns can one determine how a given message-passing version corresponds to the original sequential program.

2.5 Concluding Remarks

Starting from a sequential application, the process of parallelizing the application is quite structured: We decompose the work into tasks, assign the tasks to processes, orchestrate data access, communication and synchronization among processes, and optionally map processes to processors. For many applications, including the simple equation solver used in this chapter, the initial decomposition and assignment are similar or identical regardless of whether a shared address space or message passing programming model is used. The differences are in orchestration, particularly in the way data structures are organized and accessed and the way communication and synchronization are performed. A shared address space allows us to use the same major data structures as in a sequential program: Communication is implicit through data accesses, and the decomposition of data is not required for correctness. In the message passing case we must synthesize the logically shared data structure from per-process private data structures: Communication is explicit, explicit decomposition of data among private address spaces (processes) is necessary, and processes must be able to name one another to communicate. On the other hand,

while a shared address space program requires additional synchronization primitives separate from the loads and stores used for implicit communication, synchronization is bundled into the explicit send and receive communication in many forms of message passing. As we examine the parallelization of more complex parallel applications such as the four case studies introduced in this chapter, we will understand the implications of these differences for ease of programming and for performance.

The parallel versions of the simple equation solver that we described here were purely to illustrate programming primitives. While we did not use versions that clearly will perform terribly (e.g. we reduced communication by using a block rather than cyclic assignment of rows, and we reduced both communication and synchronization dramatically by first accumulating into local `mydiffs` and only then into a global `diff`), they can use improvement. We shall see how in the next chapter, as we turn our attention to the performance issues in parallel programming and how positions taken on these issues affect the workload presented to the architecture.

2.6 References

[GP90]   Green, S.A. and Paddon, D.J. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, vol. 6, 1990, pp. 62-73.

[KLS+94]   C. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, The High Performance Fortran Handbook. MIT Press, 1994.

[HPF93   High Performance Fortran Forum. High Performance Fortran Language Specification. Scientific Programming, vol. 2, pp. 1-270, 1993.

[Hil85]   Hillis, W.D. The Connection Machine. MIT Press, 1985.

[HiS86]   Hillis, W.D. and Steele, G.L. Data Parallel Algorithms. Communications of the ACM, 29(12), pp. 1170-1183, December 1986.

[KG+94]   Kumar, V., Gupta, A., Grama, A. and Karypis, G. Parallel Computing. <<xxx>>

[LO+87]   Lusk, E.W., Overbeek, R. et al. Portable Programs for Parallel Processors. Holt, Rinehart and Winston, Inc. 1987.

[MPI93]   Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report CS-93-214, Computer Science Department, University of Tennessee, Knoxville, November 1993.

[Pac96]   Pacheco, Peter. Parallel Programming with MPI. Morgan Kaufman Publishers, 1996.

[Pie88]   Pierce, Paul. The NX/2 Operating System. Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pp. 384--390, January, 1988.

[GLS94]   Gropp, W., Lusk, E. and Skjellum, A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, 1994.

[RSL93]   Rinard, M.C., Scales, D.J. and Lam, M.S. Jade: A High-Level, Machine-Independent Language for Parallel Programming. IEEE Computer, June 1993.

[SGL94]   Singh, J.P., Gupta, A. and Levoy, M. Parallel Visualization Algorithms: Performance and Architectural Implications. IEEE Computer, vol. 27, no. 6, June 1994.

[SH+95]   Singh, J.P., Holt, C., Totsuka, T., Gupta, A. and Hennessy, J.L. Load balancing and data locality in Hierarchial N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. Journal of Parallel and Distributed Computing, 1995 <<complete citation>>.

[SWG92]   Singh, J.P., Weber, W-D., and Gupta, A. SPLASH: The Stanford ParalleL Applications for Shared

Memory. Computer Architecture News, vol. 20, no. 1, pp. 5-44, March 1992.

2.7 Exercises

2.1 Short Answer Questions:

a. Describe two examples where the a good parallel algorithm must be based on a serial algorithm that is different than the best serial algorithm, since the latter does not afford enough concurrency.

b. Which of the case study applications that we have described (Ocean, Barnes-Hut, Ray-trace) do you think are amenable to decomposing data rather than computation and using an owner computes rule in parallelization? What do you think would be the problem(s) with using a strict data distribution and owner computes rule in the others?

2.2 <2.4.5>There are two dominant models for how parent and children processes relate to each other in a shared address space. In the heavyweight UNIX process fork model, when a process creates another, the child gets a private *copy* of the parent's image: that is, if the parent had allocated a variable x, then the child also finds a variable x in its address space which is initialized to the value that the parent had for x when it created the child; however, any modifications that either process makes subsequently are to its own copy of x and are not visible to the other process. In the lightweight threads model, the child process or thread gets a pointer to the parent's image, so that it and the parent now see the same storage location for x. All data that any process or thread allocates are shared in this model, except those that are on a procedure's stack.

a. Consider the problem of a process having to reference its process identifier procid in various parts of a program, in different routines in a call chain from the routine at which the process begins execution. How would you implement this in the first case? In the second? Do you need private data per process, or could you do this with all data being globally shared?

b. A program written in the former (fork) model may rely on the fact that a child process gets its own private copies of the parents' data structures. What changes would you make to port the program to the latter (threads) model for data structures that are (i) only read by processes after the creation of the child, (ii) are both read and written? What performance issues might arise in designing the data structures (you will be better equipped to answer this part of the question after reading the next few chapters)?

2.3 Synchronization.

a. The classic bounded buffer problem provides an example of point-to-point event synchronization. Two processes communicate through a finite buffer. One process—the producer—adds data items to a buffer when it is not full, and another—the consumer—reads data items from the buffer when it is not empty. If the consumer finds the buffer empty, it must wait till the producer inserts an item. When the producer is ready to insert an item, it checks to see if the buffer is full, in which case it must wait till the consumer removes something from the buffer. If the buffer is empty when the producer tries to add an item, depending on the implementation the consumer may be waiting for notification, so the producer may need to notify the consumer. Can you implement a bounded buffer with only point-to-point event synchronization, or do you need mutual exclusion as well. Design an implementation, including pseudocode.

    b. Why wouldn't we use spinning for interprocess synchronization in uniprocessor operating systems? What do you think are the tradeoffs between blocking and spinning on a multi-processor?

    c. In the shared address space parallel equation solver (Figure 2-13 on page 114), why do we need the second barrier at the end of a while loop iteration (line 25f)? Can you eliminate it without inserting any other synchronization, but perhaps modifying when certain operations are performed? Think about all possible scenarios.

2.4 Do LU factorization as an exercise. Describe, give a simple contiguous decomposition and assignment, draw concurrency profile, estimate speedup assuming no communication overheads, and ask to implement in a shared address space and in message passing. Modify the message-passing pseudocode to implement an interleaved assignment. Use both synchronous and asynchronous (blocking) sends and receives. Then interleaving in both directions, do the same.

2.5 More synchronization. Suppose that a system supporting a shared address space did not support barriers but only semaphores. Event synchronization would have to be constructed through semaphores or ordinary flags. To coordinate P1 indicating to P2 that it has reached point a (so that P2 can proceed past point b where it was waiting) using semaphores, *P1* performs a *wait* (also called *P* or *down*) operation on a semaphore when it reaches point *a*, and *P2* performs a *signal* (or *V* or *up*) operation on the same semaphore when it reaches point *b*. If *P1* gets to *a* before *P2* gets to *b*, *P1* suspends itself and is awoken by *P2*'s signal operation.

    a. How might you orchestrate the synchronization in LU factorization with (i) flags and (ii) semaphores replacing the barriers. Could you use point-to-point or group event synchronization instead of global event synchronization?

    b. Answer the same for the equation solver example.

2.6 Other than the above "broadcast" approach, LU factorization can also be parallelized in a form that is more aggressive in exploiting the available concurrency. We call this form the *pipelined* form of parallelization, since an element is computed from the producing process to a consumer in pipelined form via other consumers, which use the element as they communicate it in the pipeline.

    a. Write shared-address-space pseudocode, at a similar level of detail as Figure 2-13 on page 114, for a version that implements pipelined parallelism at the granularity of individual elements (as described briefly in Section 2.4.2). Show all synchronization necessary. Do you need barriers?

    b. Write message-passing pseudocode at the level of detail of Figure 2-16 on page 120 for the above pipelined case. Assume that the only communication primitives you have are synchronous and asynchronous (blocking and nonblocking) sends and receives. Which versions of send and receive would you use, and why wouldn't you choose the others?

    c. Discuss the tradeoffs (programming difficulty and likely performance differences) in programming the shared address space and message-passing versions.

    d. Discuss the tradeoffs in programming the loop-based versus pipelined parallelism

2.7 Multicast (sending a message from one process to a named list of other processes) is a useful mechanism for communicating among subsets of processes.

    a. How would you implement the message-passing, interleaved assignment version of LU factorization with multicast rather than broadcast? Write pseudocode, and compare the programming ease of the two versions.

    b. Which do you think will perform better and why?

c. What other "group communication" primitives other than multicast do you think might be useful for a message passing system to support? Give examples of computations in which they might be used.

CHAPTER 3 Programming for Performance

3.1 Introduction

The goal of using multiprocessors is to obtain high performance. Having understood concretely how the decomposition, assignment and orchestration of a parallel program are incorporated in the code that runs on the machine, we are ready to examine the key factors that limit parallel performance, and understand how they are addressed in a wide range of problems. We will see how decisions made in different steps of the programming process affect the runtime characteristics presented to the architecture, as well as how the characteristics of the architecture influence programming decisions. Understanding programming techniques and these interdependencies is important not just for parallel software designers but also for architects. Besides helping us understand parallel programs as workloads for the systems we build, it also helps us appreciate hardware-software tradeoffs; that is, in what aspects of programmability and performance can the architecture be of assistance, and what aspects are best left to software. The interdependencies of program and system are more fluid, more complex and have far greater performance impact in

multiprocessors than in uniprocessors; hence, this understanding is very important to our goal of designing high-performance systems that reduce cost and programming effort. We will carry it forward with us throughout the book, starting with concrete guidelines for workload-driven architectural evaluation in the next chapter.

The space of performance issues and techniques in parallel software is very rich: Different goals trade off with one another, and techniques that further one goal may cause us to revisit the techniques used to address another. This is what makes the creation of parallel software so interesting. As in uniprocessors, most performance issues can be addressed either by algorithmic and programming techniques in software or by architectural techniques or both. The focus of this chapter is on the issues and on software techniques. Architectural techniques, sometimes hinted at here, are the subject of the rest of the book.

While there are several interacting performance issues to contend with, they are not all dealt with at once. The process of creating a high-performance program is one of successive refinement. As discussed in Chapter 2, the partitioning steps—decomposition and assignment—are largely independent of the underlying architecture or communication abstraction, and concern themselves with major algorithmic issues that depend only on the inherent properties of the problem. In particular, these steps view the multiprocessor as simply a set of processors that communicate with one another. Their goal is to resolve the tension between balancing the workload across processes, reducing interprocess communication, and reducing the extra work needed to compute and manage the partitioning. We focus our attention first on addressing these partitioning issues.

Next, we open up the architecture and examine the new performance issues it raises for the orchestration and mapping steps. Opening up the architecture means recognizing two facts. The first fact is that a multiprocessor is not only a collection of processors but also a collection of memories, one that an individual processor can view as an extended memory hierarchy. The management of data in these memory hierarchies can cause more data to be transferred across the network than the inherent communication mandated by the partitioning of the work among processes in the parallel program. The actual communication that occurs therefore depends not only on the partitioning but also on how the program's access patterns and locality of data reference interact with the organization and management of the extended memory hierarchy. The second fact is that the cost of communication as seen by the processor—and hence the contribution of communication to the execution time of the program—depends not only on the amount of communication but also on how it is structured to interact with the architecture. The relationship between communication, data locality and the extended memory hierarchy is discussed in Section 3.3. Then, Section 3.4 examines the software techniques to address the major performance issues in orchestration and mapping: techniques for reducing the extra communication by exploiting data locality in the extended memory hierarchy, and techniques for structuring communication to reduce its cost.

Of course, the architectural interactions and communication costs that we must deal with in orchestration sometimes cause us to go back and revise our partitioning methods, which is an important part of the refinement in parallel programming. While there are interactions and tradeoffs among all the performance issues we discuss, the chapter discusses each independently as far as possible and identifies tradeoffs as they are encountered. Examples are drawn from the four case study applications throughout, and the impact of some individual programming techniques illustrated through measurements on a particular cache-coherent machine with physically distributed memory, the Silicon Graphics Origin2000, which is described in detail in Chapter 8.The equation solver kernel is aso carried through the discussion, and the performance

techniques are applied to it as relevant, so by the end of the discussion we will have created a high-performance parallel version of the solver.

As we go through the discussion of performance issues, we will develop simple analytical models for the speedup of a parallel program, and illustrate how each performance issue affects the speedup equation. However, from an architectural perspective, a more concrete way of looking at performance is to examine the different components of execution time as seen by an individual processor in a machine; i.e. how much time the processor spends executing instructions, accessing data in the extended memory hierarchy, and waiting for synchronization events to occur. In fact, these components of execution time can be mapped directly to the performance factors that software must address in the steps of creating a parallel program. Examining this view of performance helps us understand very concretely what a parallel execution looks like as a workload presented to the architecture, and the mapping helps us understand how programming techniques can alter this profile. Together, they will help us learn how to use programs to evaluate architectural tradeoffs in the next chapter. This view and mapping are discussed in Section 3.5.

Once we have understood the performance issues and techniques, we will be ready to do what we wanted to all along: understand how to create high-performance parallel versions of complex, realistic applications; namely, the four case studies. Section 3.6 applies the parallelization process and the performance techniques to each case study in turn, illustrating how the techniques are employed together and the range of resulting execution characteristics presented to an architecture, reflected in varying profiles of execution time. We will also finally be ready to fully understand the implications of realistic applications and programming techniques for tradeoffs between the two major lower-level programming models: a shared address space and explicit message passing. The tradeoffs of interest are both in ease of programming and in performance, and will be discussed in Section 3.7. Let us begin with the algorithmic performance issues in the decomposition and assignment steps.

3.2 Partitioning for Performance

For these steps, we can view the machine as simply a set of cooperating processors, largely ignoring its programming model and organization. All we know at this stage is that communication between processors is expensive. The three primary algorithmic issues are:

- *balancing* the workload and reducing the time spent waiting at synchronization events
- reducing *communication*, and
- reducing the *extra work* done to determine and manage a good assignment.

Unfortunately, even the three primary algorithmic goals are at odds with one another and must be traded off. A singular goal of minimizing communication would be satisfied by running the program on a single processor, as long as the necessary data fit in the local memory, but this would yield the ultimate load imbalance. On the other hand, near perfect load balance could be achieved, at a tremendous communication and task management penalty, by making each primitive operation in the program a task and assigning tasks randomly. And in many complex applications load balance and communication can be improved by spending more time determining a good assignment (extra work). The goal of decomposition and assignment is to achieve a good compromise between these conflicting demands. Fortunately, success is often not so difficult in practice, as we shall see. Throughout our discussion of performance issues, the four case studies

from Chapter 2 will be used to illustrate issues and techniques. They will later be presented as complete case studies addressing all the performance issues in Section 3.6. For each issue, we will also see how it is applied to the simple equation solver kernel from Chapter 2, resulting at the end in a high performance version.

3.2.1 Load Balance and Synchronization Wait Time

In its simplest form, balancing the workload means ensuring that every processor does the same amount of work. It extends exposing enough concurrency—which we saw earlier—with proper assignment and reduced serialization, and gives the following simple limit on potential speedup:

$$Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{max \text{ Work on any Processor}}$$

Work, in this context, should be interpreted liberally, because what matters is not just how many calculations are done but the time spent doing them, which involves data accesses and communication as well.

In fact, load balancing is a little more complicated than simply equalizing work. Not only should different processors do the same amount of work, but they should be working at the same time. The extreme point would be if the work were evenly divided among processes but only one process were active at a time, so there would be no speedup at all! The real goal of load balance is to minimize the time processes spend waiting at synchronization points, including an implicit one at the end of the program. This also involves minimizing the serialization of processes due to either mutual exclusion (waiting to enter critical sections) or dependences. The assignment step should ensure that reduced serialization is possible, and orchestration should ensure that it happens.

There are four parts to balancing the workload and reducing synchronization wait time:

- Identifying enough concurrency in decomposition, and overcoming Amdahl's Law.
- Deciding how to manage the concurrency (statically or dynamically).
- Determining the granularity at which to exploit the concurrency.
- Reducing serialization and synchronization cost.

This section examines some techniques for each, using examples from the four case studies and other applications as well.

Identifying Enough Concurrency: Data and Function Parallelism

We saw in the equation solver that concurrency may be found by examining the loops of a program or by looking more deeply at the fundamental dependences. Parallelizing loops usually leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure(s). This is called *data parallelism*, and is a more general form of the parallelism that inspired data parallel architectures discussed in Chapter 1. Computing forces on different particles in Barnes-Hut is another example.

In addition to data parallelism, applications often exhibit *function parallelism* as well: Entirely different calculations are performed concurrently on either the same or different data. *Function parallelism* is often referred to as control parallelism or task parallelism, though these are overloaded terms. For example, setting up an equation system for the solver in Ocean requires many

different computations on ocean cross-sections, each using a few cross-sectional grids. Analyzing dependences at the grid or array level reveals that several of these computations are independent of one another and can be performed in parallel. Pipelining is another form of function parallelism in which different sub-operations or stages of the pipeline are performed concurrently. In encoding a sequence of video frames, each block of each frame passes through several stages: pre-filtering, convolution from the time to the frequency domain, quantization, entropy coding, etc. There is pipeline parallelism across these stages (for example a few processes could be assigned to each stage and operate concurrently), as well as data parallelism between frames, among blocks in a frame, and within an operation on a block.

Function parallelism and data parallelism are often available together in an application, and provide a hierarchy of levels of parallelism from which we must choose (e.g. function parallelism across grid computations and data parallelism within grid computations in Ocean). Orthogonal levels of parallelism are found in many other applications as well; for example, applications that route wires in VLSI circuits exhibit parallelism across the wires to be routed, across the segments within a wire, and across the many routes evaluated for each segment (see Figure 3-1).
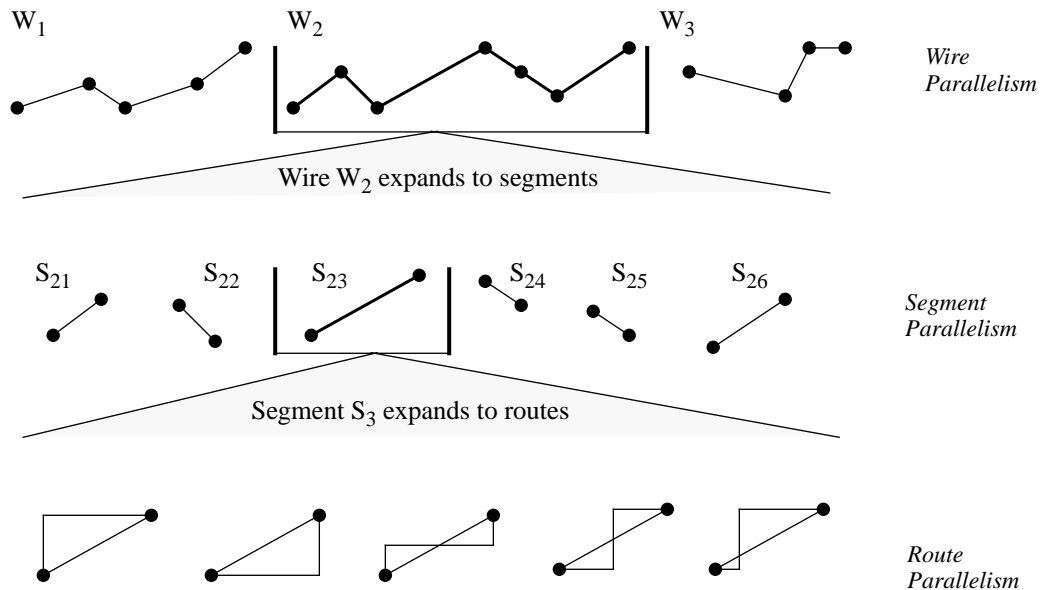
Figure 3-1 The three axes of parallelism in a VLSI wire routing application.

The degree of available function parallelism is usually modest and does not grow much with the size of the problem being solved. The degree of data parallelism, on the other hand, usually grows with data set size. Function parallelism is also often more difficult to exploit in a load balanced way, since different functions involve different amounts of work. Most parallel programs that run on large-scale machines are data parallel according to our loose definition of the term, and function parallelism is used mainly to reduce the amount of global synchronization required between data parallel computations (as illustrated in Ocean, in Section 3.6.1).

By identifying the different types of concurrency in an application we often find much more than we need. The next step in decomposition is to restrict the available concurrency by determining

the granularity of tasks. However, the choice of task size also depends on how we expect to manage the concurrency, so we discuss this next.

Determining How to Manage Concurrency: Static versus Dynamic Assignment

A key issue in exploiting concurrency is whether a good load balance can be obtained by a static or predetermined assignment, introduced in the previous chapter, or whether more dynamic means are required. A static assignment is typically an algorithmic mapping of tasks to processes, as in the simple equation solver kernel discussed in the previous chapter. Exactly which tasks (grid points or rows) are assigned to which processes may depend on the problem size, the number of processes, and other parameters, but the assignment does not adapt at runtime to other environmental considerations. Since the assignment is predetermined, static techniques do not incur much task management overhead at runtime. However, to achieve good load balance they require that the work in each task be predictable enough, or that there be so many tasks that the statistics of large numbers ensure a balanced distribution, with pseudo-random assignment. In addition to the program itself, it is also important that other environmental conditions such as interference from other applications not perturb the relationships among processors limiting the robustness of static load balancing.

Dynamic techniques adapt to load imbalances at runtime. They come in two forms. In *semi-static* techniques the assignment for a phase of computation is determined algorithmically before that phase, but assignments are recomputed periodically to restore load balance based on profiles of the actual workload distribution gathered at runtime. That is, we profile (measure) the work that each task does in one phase, and use that as an estimate of the work associated with it in the next phase. This repartitioning technique is used to assign stars to processes in Barnes-Hut (Section 3.6.2), by recomputing the assignment between time-steps of the galaxy's evolution. The galaxy evolves slowly, so the workload distribution among stars does not change much between successive time-steps. Figure 3-2(a) illustrates the advantage of semi-static partitioning
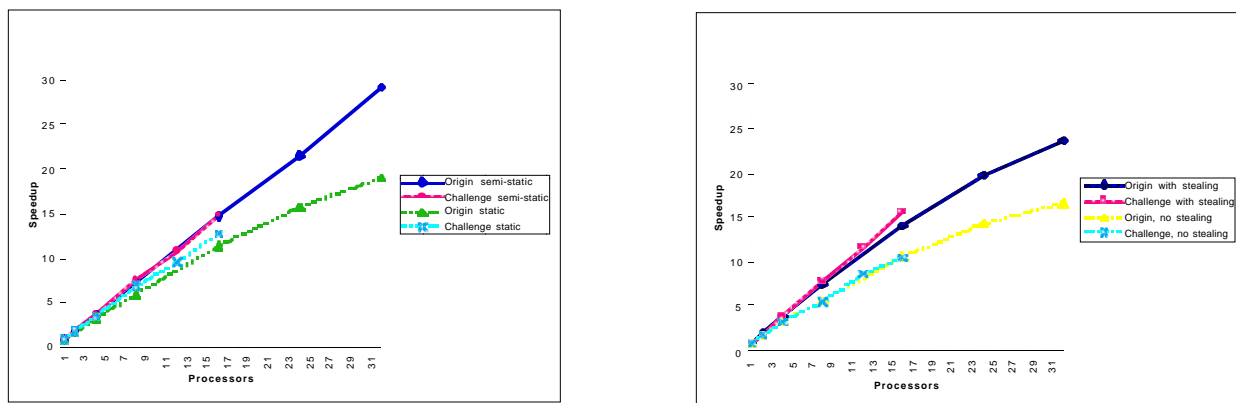
Figure 3-2 Illustration of the performance impact of dynamic partitioning for load balance.

The graph on the left shows the speedups of the Barnes-Hut application with and without semi-static partitioning, and the graph on the right shows the speedups of Raytrace with and without task stealing. Even in these applications that have a lot of parallelism, dynamic partitioning is important for improving load balance over static partitioning.

over a static assignment of particles to processors, for a 512K particle execution measured on the Origin2000. It is clear that the performance difference grows with the number of processors used.

The second dynamic technique, *dynamic tasking*, is used to handle the cases where either the work distribution or the system environment is too unpredictable even to periodically recompute a load balanced assignment.[1] For example, in Raytrace the work associated with each ray is impossible to predict, and even if the rendering is repeated from different viewpoints the change in viewpoints may not be gradual. The dynamic tasking approach divides the computation into tasks and maintains a pool of available tasks. Each process repeatedly takes a task from the pool and executes it—possibly inserting new tasks into the pool—until there are no tasks left. Of course, the management of the task pool must preserve the dependences among tasks, for example by inserting a task only when it is ready for execution. Since dynamic tasking is widely used, let us look at some specific techniques to implement the task pool. Figure 3-2(b) illustrates the advantage of dynamic tasking over a static assignment of rays to processors in the Raytrace application, for the balls data set measured on the Origin2000.

A simple example of dynamic tasking in a shared address space is *self-scheduling* of a parallel loop. The loop counter is a shared variable accessed by all the processes that execute iterations of the loop. Processes obtain a loop iteration by incrementing the counter (atomically), execute the iteration, and access the counter again, until no iterations remain. The task size can be increased by taking multiple iterations at a time, i.e., adding a larger value to the shared loop counter. In *guided self-scheduling* [AiN88] processes start by taking large chunks and taper down the chunk size as the loop progresses, hoping to reduce the number of accesses to the shared counter without compromising load balance.

More general dynamic task pools are usually implemented by a collection of queues, into which tasks are inserted and from which tasks are removed and executed by processes. This may be a single centralized queue or a set of distributed queues, typically one per process, as shown in Figure 3-3. A centralized queue is simpler, but has the disadvantage that every process accesses the same task queue, potentially increasing communication and causing processors to contend for queue access. Modifications to the queue (enqueuing or dequeuing tasks) must be mutually exclusive, further increasing contention and causing serialization. Unless tasks are large and there are few queue accesses relative to computation, a centralized queue can quickly become a performance bottleneck as the number of processors increases.

With distributed queues, every process is initially assigned a set of tasks in its local queue. This initial assignment may be done intelligently to reduce interprocess communication providing more control than self-scheduling and centralized queues. A process removes and executes tasks from its local queue as far as possible. If it creates tasks it inserts them in its local queue. When there are no more tasks in its local queue it queries other processes' queues to obtain tasks from them, a mechanism known as *task stealing*. Because task stealing implies communication and can generate contention, several interesting issues arise in implementing stealing; for example,

---

1. The applicability of static or semi-static assignment depends not only on the computational properties of the program but also on its interactions with the memory and communication systems and on the predictability of the execution environment. For example, differences in memory or communication stall time (due to cache misses, page faults or contention) can cause imbalances observed at synchronization points even when the workload is computationally load balanced. Static assignment may also not be appropriate for time-shared or heterogeneous systems.

(a) Centralized task queue   (b) Distributed task queues (one per process)
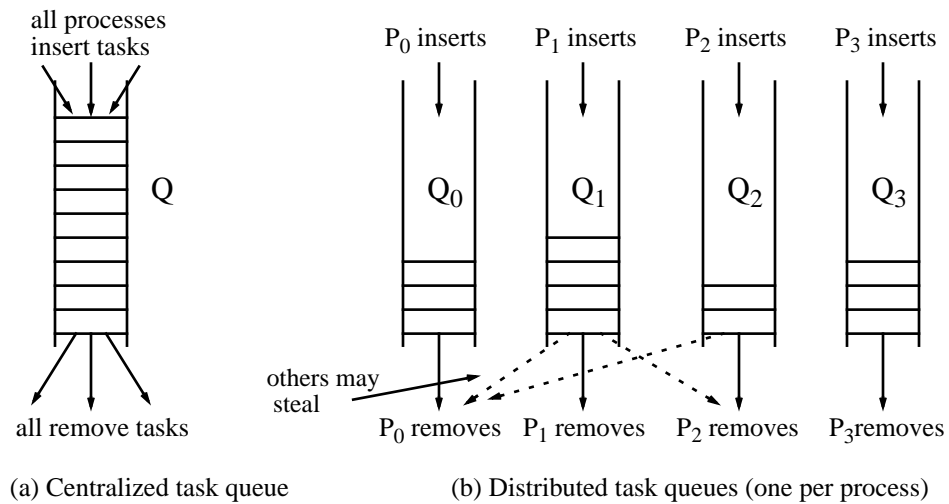
Figure 3-3 Implementing a dynamic task pool with a system of task queues.

how to minimize stealing, whom to steal from, how many and which tasks to steal at a time, and so on. Stealing also introduces the important issue of *termination detection*: How do we decide when to stop searching for tasks to steal and assume that they're all done, given that tasks generate other tasks that are dynamically inserted in the queues? Simple heuristic solutions to this problem work well in practice, although a robust solution can be quite subtle and communication intensive [DS68,CM88]. Task queues are used in both a shared address space, where the queues are shared data structures that are manipulated using locks, as well as with explicit message passing where the owners of queues service requests for them.

While dynamic techniques generally provide good load balancing despite unpredictability or environmental conditions, they make task management more expensive. Dynamic tasking techniques also compromise the explicit control over which tasks are executed by which processes, thus potentially increasing communication and compromising data locality. Static techniques are therefore usually preferable when they can provide good load balance given both application and environment.

Determining the Granularity of Tasks

If there are no load imbalances due to dependences among tasks (for example, if all tasks to be executed are available at the beginning of a phase of computation) then the maximum load imbalance possible with a task-queue strategy is equal to the granularity of the largest task. By *granularity* we mean the amount of work associated with a task, measured by the number of instructions or—more appropriately—the execution time. The general rule for choosing a granularity at which to actually exploit concurrency is that fine-grained or small tasks have the potential for better load balance (there are more tasks to divide among processes and hence more concurrency), but they lead to higher task management overhead, more contention and more interprocessor communication than coarse-grained or large tasks. Let us see why, first in the context of dynamic task queuing where the definitions and tradeoffs are clearer.

Task Granularity with Dynamic Task Queueing: Here, a task is explicitly defined as an entry placed on a task queue, so task granularity is the work associated with such an entry. The larger task management (queue manipulation) overhead with small tasks is clear: At least with a centralized queue, the more frequent need for queue access generally leads to greater contention as well. Finally, breaking up a task into two smaller tasks might cause the two tasks to be executed on different processors, thus increasing communication if the subtasks access the same logically shared data.

Task Granularity with Static Assignment: With static assignment, it is less clear what one should call a task or a unit of concurrency. For example, in the equation solver is a task a group of rows, a single row, or an individual element? We can think of a task as the largest unit of work such that even if the assignment of tasks to processes is changed, the code that implements a task need not change. With static assignment, task size has a much smaller effect on task management overhead compared to dynamic task-queueing, since there are no queue accesses. Communication and contention are affected by the assignment of tasks to processors, not their size. The major impact of task size is usually on load imbalance and on exploiting data locality in processor caches.

Reducing Serialization

Finally, to reduce serialization at synchronization points—whether due to mutual exclusion or dependences among tasks—we must be careful about how we assign tasks and also how we orchestrate synchronization and schedule tasks. For event synchronization, an example of excessive serialization is the use of more conservative synchronization than necessary, such as barriers instead of point-to-point or group synchronization. Even if point-to-point synchronization is used, it may preserve data dependences at a coarser grain than necessary; for example, a process waits for another to produce a whole row of a matrix when the actual dependences are at the level of individual matrix elements. However, finer-grained synchronization is often more complex to program and implies the execution of more synchronization operations (say one per word rather than one per larger data structure), the overhead of which may turn out to be more expensive than the savings in serialization. As usual, tradeoffs abound.

For mutual exclusion, we can reduce serialization by using separate locks for separate data items, and making the critical sections protected by locks smaller and less frequent if possible. Consider the former technique. In a database application, we may want to lock when we update certain fields of records that are assigned to different processes. The question is how to organize the locking. Should we use one lock per process, one per record, or one per field? The finer the granularity the lower the contention but the greater the space overhead and the less frequent the reuse of locks. An intermediate solution is to use a fixed number of locks and share them among records using a simple hashing function from records to locks. Another way to reduce serialization is to stagger the critical sections in time, i.e. arrange the computation so multiple processes do not try to access the same lock at the same time.

Implementing task queues provides an interesting example of making critical sections smaller and also less frequent. Suppose each process adds a task to the queue, then searches the queue for another task with a particular characteristic, and then remove this latter task from the queue. The task insertion and deletion may need to be mutually exclusive, but the searching of the queue does not. Thus, instead of using a single critical section for the whole sequence of operations, we

can break it up into two critical sections (insertion and deletion) and non-mutually-exclusive code to search the list in between.

More generally, checking (reading) the state of a protected data structure usually does not have to be done with mutual exclusion, only modifying the data structure does. If the common case is to check but not have to modify, as for the tasks we search through in the task queue, we can check without locking, and then lock and re-check (to ensure the state hasn't changed) within the critical section only if the first check returns the appropriate condition. Also, instead of using a single lock for the entire queue, we can use a lock per queue element so elements in different parts of the queue can be inserted or deleted in parallel (without serialization). As with event synchronization, the correct tradeoffs in performance and programming ease depend on the costs and benefits of the choices on a system.

We can extend our simple limit on speedup to reflect both load imbalance and time spent waiting at synchronization points as follows:

$$Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{max(\text{Work} + \text{Synch Wait Time})}$$

In general, the different aspects of balancing the workload are the responsibility of software: There is not much an architecture can do about a program that does not have enough concurrency or is not load balanced. However, an architecture can help in some ways. First, it can provide efficient support for load balancing techniques such as task stealing that are used widely by parallel software (applications, libraries and operating systems). Task queues have two important architectural implications. First, an access to a remote task queue is usually a probe or query, involving a small amount of data transfer and perhaps mutual exclusion. The more efficiently fine-grained communication and low-overhead, mutually exclusive access to data are supported, the smaller we can make our tasks and thus improve load balance. Second, the architecture can make it easy to name or access the logically shared data that a stolen task needs. Third, the architecture can provide efficient support for point-to-point synchronization, so there is more incentive to use them instead of conservative barriers and hence achieve better load balance.

3.2.2 Reducing Inherent Communication

Load balancing by itself is conceptually quite easy as long as the application affords enough concurrency: We can simply make tasks small and use dynamic tasking. Perhaps the most important tradeoff with load balance is reducing interprocessor communication. Decomposing a problem into multiple tasks usually means that there will be communication among tasks. If these tasks are assigned to different processes, we incur communication among processes and hence processors. We focus here on reducing communication that is *inherent* to the parallel program—i.e. one process produces data that another needs—while still preserving load balance, retaining our view of the machine as a set of cooperating processors. We will see in Section 3.3 that in a real system communication occurs for other reasons as well.

The impact of communication is best estimated not by the absolute amount of communication but by a quantity called the *communication to computation ratio*. This is defined as the amount of communication (in bytes, say) divided by the computation time, or by the number of instructions executed. For example, a gigabyte of communication has a much greater impact on the execution time and communication bandwidth requirements of an application if the time required for the

application to execute is 1 second than if it is 1 hour! The communication to computation ratio may be computed as a per-process number, or accumulated over all processes.

The inherent communication to computation ratio is primarily controlled by the assignment of tasks to processes. To reduce communication, we should try to ensure that tasks that access the same data or need to communicate a lot are assigned to the same process. For example, in a database application, communication would be reduced if queries and updates that access the same database records are assigned to the same process.

One partitioning principle that has worked very well for load balancing and communication volume in practice is *domain decomposition*. It was initially used in data-parallel scientific computations such as Ocean, but has since been found applicable to many other areas. If the data set on which the application operates can be viewed as a physical domain—for example, the grid representing an ocean cross-section in Ocean, the space containing a galaxy in Barnes-Hut, or the image for graphics or video applications—then it is often the case that a point in the domain either requires information directly from only a small localized region around that point, or requires long-range information but the requirements fall off with distance from the point. We saw an example of the latter in Barnes-Hut. For the former, algorithms for motion estimation in video encoding and decoding examine only areas of a scene that are close to the current pixel, and a point in the equation solver kernel needs to access only its four nearest neighbor points directly. The goal of partitioning in these cases is to give every process a contiguous region of the domain while of course retaining load balance, and to shape the domain so that most of the process's information requirements are satisfied within its assigned partition. As Figure 3-4 shows, in many such cases the communication requirements for a process grow proportionally to the size of a partition's boundary, while computation grows proportionally to the size of its entire partition. The communication to computation ratio is thus a surface area to volume ratio in three dimensions, and perimeter to area in two dimensions. Like load balance in data parallel computation, it can be reduced by either increasing the data set size (n^2 in the figure) or reducing the number of processors (p).
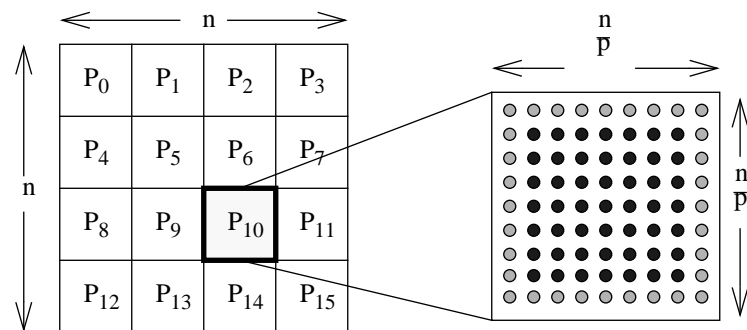
Figure 3-4 The perimeter to area relationship of communication to computation in a two-dimensional domain decomposition.

The example shown is for an algorithm with localized, nearest-neighbor information exchange like the simple equation solver kernel. Every point on the grid needs information from its four nearest neighbors. Thus, the darker, internal points in processor P_{10}'s partition do not need to communicate directly with any points outside the partition. Computation for processor P_{10} is thus proportional to the sum of all $\frac{n^2}{p}$ points, while communication is proportional to the number of lighter, boundary points, which is $4\frac{n}{\sqrt{p}}$.

Of course, the ideal shape for a domain decomposition is application-dependent, depending primarily on the information requirements of and work associated with the points in the domain. For the equation solver kernel in Chapter 2, we chose to partition the grid into blocks of contiguous rows. Figure 3-5 shows that partitioning the grid into square-like subgrids leads to a lower inherent communication-to-computation ratio. The impact becomes greater as the number of processors increases relative to the grid size. We shall therefore carry forward this partitioning into square subgrids (or simply "subgrids") as we continue to discuss performance. As a simple exer-
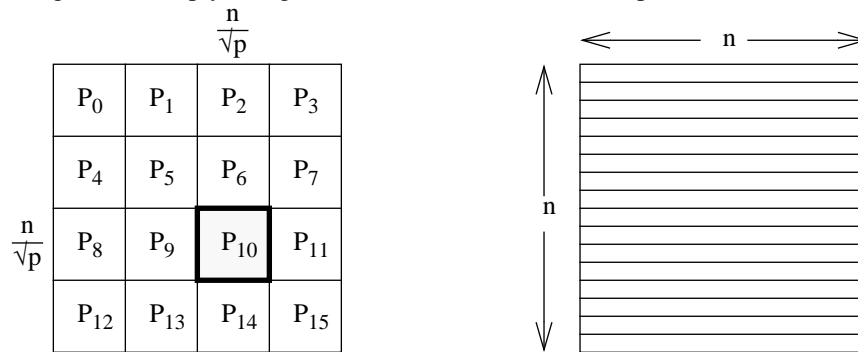
Figure 3-5 Choosing among domain decompositions for a simple nearest-neighbor computation on a regular two-dimensional grid.

Since the work per grid point is uniform, equally sized partitions yield good load balance. But we still have choices. We might partition the elements of the grid into either strips of contiguous rows (right) or block-structured partitions that are as close to square as possible (left). The perimeter-to-area (and hence communication to computation) ratio in the block decomposition case is $\dfrac{4 \times n/\sqrt{p}}{n^2/p}$

or $\dfrac{4 \times \sqrt{p}}{n}$, while that in strip decomposition is $\dfrac{2 \times n}{n^2/p}$ or $\dfrac{2 \times p}{n}$. As p increases, block decomposition incurs less inherent communication for the same computation than strip decomposition.

cise, think about what the communication to computation ratio would be if we assigned rows to processes in an interleaved or cyclic fashion instead (row i assigned to process i mod *nprocs*).

How do we find a suitable domain decomposition that is load balanced and also keeps communication low? There are several ways, depending on the nature and predictability of the computation:

- *Statically, by inspection*, as in the equation solver kernel and in Ocean. This requires predictability and usually leads to regularly shaped partitions, as in Figures 3-4 and 3-5.

- *Statically, by analysis.* The computation and communication characteristics may depend on the input presented to the program at runtime, thus requiring an analysis of the input. However, the partitioning may need to be done only once after the input analysis—before the actual computation starts—so we still consider it static. Partitioning sparse matrix computations used in aerospace and automobile simulations is an example: The matrix structure is fixed, but is highly irregular and requires sophisticated graph partitioning. In data mining, we may divide the database of transactions statically among processors, but a balanced assignment of itemsets to processes requires some analysis since the work associated with different itemsets is not equal. A simple static assignment of itemsets as well as the database by inspection keeps communication low, but does not provide load balance.

- *Semi-statically* (with periodic repartitioning). This was discussed earlier for applications like Barnes-Hut whose characteristics change with time but slowly. Domain decomposition is still important to reduce communication, and we will see the profiling-based Barnes-Hut example in Section 3.6.2.

- *Statically or semi-statically, with dynamic task stealing*. Even when the computation is highly unpredictable and dynamic task stealing must be used, domain decomposition may be useful in initially assigning tasks to processes. Raytrace is an example. Here there are two domains: the three-dimensional scene being rendered, and the two-dimensional image plane. Since the natural tasks are rays shot through the image plane, it is much easier to manage domain decomposition of that plane than of the scene itself. We decompose the image domain just like the grid in the equation solver kernel (Figure 3-4), with image pixels corresponding to grid points, and initially assign rays to the corresponding processes. Processes then steal rays (pixels) or group of rays for load balancing. The initial domain decomposition is useful because rays shot through adjacent pixels tend to access much of the same scene data.

Of course, partitioning into a contiguous subdomain per processor is not always appropriate for high performance in all applications. We shall see examples in matrix factorization in Exercise 3.3, and in a radiosity application from computer graphics in Chapter 4. Different phases of the same application may also call for different decompositions. The range of techniques is very large, but common principles like domain decomposition can be found. For example, even when stealing tasks for load balancing in very dynamic applications, we can reduce communication by searching other queues in the same order every time, or by preferentially stealing large tasks or several tasks at once to reduce the number of times we have to access non-local queues.

In addition to reducing communication volume, it is also important to keep communication balanced among processors, not just computation. Since communication is expensive, imbalances in communication can translate directly to imbalances in execution time among processors. Overall, whether tradeoffs should be resolved in favor of load balance or communication volume depends on the cost of communication on a given system. Including communication as an explicit performance cost refines our basic speedup limit to:

$$Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{max(\text{Work} + \text{Synch. Wait Time} + \text{Comm. Cost})}$$

What we have done in this expression is separated out communication from work. Work now means instructions executed plus local data access costs.

The amount of communication in parallel programs clearly has important implications for architecture. In fact, architects examine the needs of applications to determine what latencies and bandwidths of communication are worth spending extra money for (see Exercise 3.8); for example, the bandwidth provided by a machine can usually be increased by throwing hardware (and hence money) at the problem, but this is only worthwhile if applications will exercise the increased bandwidth. As architects, we assume that the programs delivered to us are reasonable in their load balance and their communication demands, and strive to make them perform better by providing the necessary support. Let us now examine the last of the algorithmic issues that we can resolve in partitioning itself, without much attention to the underlying architecture.

3.2.3 Reducing the Extra Work

The discussion of domain decomposition above shows that when a computation is irregular, computing a good assignment that both provides load balance and reduces communication can be quite expensive. This extra work is not required in a sequential execution, and is an overhead of parallelism. Consider the sparse matrix example discussed above as an example of static partitioning by analysis. The matrix can be represented as a graph, such that each node represents a row or column of the matrix and an edge exists between two nodes i and j if the matrix entry (i,j) is nonzero. The goal in partitioning is to assign each process a set of nodes such that the number of edges that cross partition boundaries is minimized and the computation is also load balanced. Many techniques have been developed for this; the ones that result in a better balance between load balance and communication require more time to compute the graph partitioning. We shall see another example in the Barnes-Hut case study in Section 3.6.2.

Another example of extra work is computing data values redundantly rather than having one process compute them and communicate them to the others, which may be a favorable tradeoff when the cost of communication is high. Examples include all processes computing their own copy of the same shading table in computer graphics applications, or of trigonometric tables in scientific computations. If the redundant computation can be performed while the processor is otherwise idle due to load imbalance, its cost can be hidden.

Finally, many aspects of orchestrating parallel programs—such as creating processes, managing dynamic tasking, distributing code and data throughout the machine, executing synchronization operations and parallelism control instructions, structuring communication appropriately for a machine, packing/unpacking data to and from communication messages—involve extra work as well. For example, the cost of creating processes is what causes us to create them once up front and have them execute tasks until the program terminates, rather than have processes be created and terminated as parallel sections of code are encountered by a single main thread of computation (a *fork-join* approach, which is sometimes used with lightweight threads instead of processes). In Data Mining, we will see that substantial extra work done to transform the database pays off in reducing communication, synchronization, and expensive input/output activity.

We must consider the tradeoffs between extra work, load balance, and communication carefully when making our partitioning decisions. The architecture can help reduce extra work by making communication and task management more efficient, and hence reducing the need for extra work. Based only on these algorithmic issues, the speedup limit can now be refined to:

$$Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{max \text{ (Work + Synch Wait Time + Comm Cost + Extra Work)}} \qquad \textbf{(EQ 3.1)}$$

3.2.4 Summary

The analysis of parallel algorithms requires a characterization of a multiprocessor and a characterization of the parallel algorithm. Historically, the analysis of parallel algorithms has focussed on algorithmic aspects like partitioning, and has not taken architectural interactions into account. In fact, the most common model used to characterize a multiprocessor for algorithm analysis has been the Parallel Random Access Memory (PRAM) model [FoW78]. In its most basic form, the PRAM model assumes that data access is free, regardless of whether it is local or involves com-

munication. That is, communication cost is zero in the above speedup expression (Equation 3.1), and work is treated simply as instructions executed:

$$S\text{peedup-PRAM}_{problem}(p) \leq \frac{\text{Sequential Instructions}}{max \ (\text{Instructions} \ + \ \text{Synch Wait Time} + \text{Extra Instructions})} \ . \quad \textbf{(EQ 3.2)}$$

A natural way to think of a PRAM machine is as a shared address space machine in which all data access is free. The performance factors that matter in parallel algorithm analysis using the PRAM are load balance and extra work. The goal of algorithm development for PRAMs is to expose enough concurrency so the workload may be well balanced, without needing too much extra work.

While the PRAM model is useful in discovering the concurrency available in an algorithm, which is the first step in parallelization, it is clearly unrealistic for modeling performance on real parallel systems. This is because communication, which it ignores, can easily dominate the cost of a parallel execution in modern systems. In fact, analyzing algorithms while ignoring communication can easily lead to a poor choice of decomposition and assignment, to say nothing of orchestration. More recent models have been developed to include communication costs as explicit parameters that algorithm designers can use [Val90, CKP+93]. We will return to this issue after we have obtained a better understanding of communication costs.

The treatment of communication costs in the above discussion has been limited for two reasons when it comes to dealing with real systems. First, communication inherent to the parallel program and its partitioning is not the only form of communication that is important: There may be substantial non-inherent or *artifactual* communication that is caused by interactions of the program with the architecture on which it runs. Thus, we have not yet modeled the amount of communication generated by a parallel program satisfactorily. Second, the "communication cost" term in the equations above is not only determined by the amount of communication caused, whether inherent or artifactual, but also by the costs of the basic communication operations in the machine, the structure of the communication in the program, and how the two interact. Both artifactual communication and communication structure are important performance issues that are usually addressed in the orchestration step since they are architecture-dependent. To understand them, and hence open up the communication cost term, we first need a deeper understanding of some critical interactions of architectures with software.

3.3 Data Access and Communication in a Multi-Memory System

In our discussion of partitioning, we have viewed a multiprocessor as a collection of cooperating processors. However, multiprocessor systems are also multi-memory, multi-cache systems, and the role of these components is essential to performance. The role is essential regardless of programming model, though the latter may influence what the specific performance issues are. The rest of the performance issues for parallel programs have primarily to do with accessing data in this multi-memory system, so it is useful for us to now take a different view of a multiprocessor.

3.3.1 A Multiprocessor as an Extended Memory Hierarchy

From an individual processor's perspective, we can view all the memory of the machine, including the caches of other processors, as forming levels of an extended memory hierarchy. The com-

munication architecture glues together the parts of the hierarchy that are on different nodes. Just as interactions with levels of the memory hierarchy (for example cache size, associativity, block size) can cause the transfer of more data between levels than is inherently necessary for the program in a uniprocessor system, so also interactions with the extended memory hierarchy can cause more communication (transfer of data across the network) in multiprocessors than is inherently necessary to satisfy the processes in a parallel program. Since communication is expensive, it is particularly important that we exploit data locality in the extended hierarchy, both to improve node performance and to reduce the extra communication between nodes.

Even in uniprocessor systems, a processor's performance depends heavily on the performance of the memory hierarchy. Cache effects are so important that it hardly makes sense to talk about performance without taking caches into account. We can look at the performance of a processor in terms of the time needed to complete a program, which has two components: the time the processor is busy executing instructions and the time it spends waiting for data from the memory system.

$$\text{Time}_{prog}(1) = Busy(1) + DataAccess(1) \qquad \textbf{(EQ 3.3)}$$

As architects, we often normalize this formula by dividing each term by the number of instructions and measuring time in clock cycles. We then have a convenient, machine-oriented metric of performance, cycles per instruction (CPI), which is composed of an ideal CPI plus the average number of stall cycles per instruction. On a modern microprocessor capable of issuing four instructions per cycle, dependences within the program might limit the average issue rate to, say, 2.5 instructions per cycle, or an ideal CPI of 0.4. If only 1% of these instructions cause a cache miss, and a cache miss causes the processor to stall for 80 cycles, on average, then these stalls will account for an additional 0.8 cycles per instruction. The processor will be busy doing "useful" work only one-third of its time! Of course, the other two-thirds of the time is in fact useful. It is the time spent communicating with memory to access data. Recognizing this data access cost, we may elect to optimize either the program or the machine to perform the data access more efficiently. For example, we may change how we access the data to enhance temporal or spatial locality or we might provide a bigger cache or latency tolerance mechanisms.

An idealized view of this extended hierarchy would be a hierarchy of local caches connected to a single "remote" memory at the next level. In reality the picture is a bit more complex. Even on machines with centralized shared memories, beyond the local caches is a multi-banked memory as well as the caches of other processors. With physically distributed memories, a part of the main memory too is local, a larger part is remote, and what is remote to one processor is local to another. The difference in programming models reflects a difference in how certain levels of the hierarchy are managed. We take for granted that the registers in the processor are managed explicitly, but by the compiler. We also take for granted that the first couple of levels of caches are managed transparently by the hardware. In the shared address space model, data movement between the remote level and the local node is managed transparently as well. The message passing model has this movement managed explicitly by the program. Regardless of the management, levels of the hierarchy that are closer to the processor provide higher bandwidth and lower latency access to data. We can improve data access performance either by improving the architecture of the extended memory hierarchy, or by improving the locality in the program.

Exploiting locality exposes a tradeoff with parallelism similar to reducing communication. Parallelism may cause more processors to access and draw toward them the same data, while locality

from a processor's perspective desires that data stay close to it. A high performance parallel program needs to obtain performance out of each individual processor—by exploiting locality in the extended memory hierarchy—as well as being well-parallelized.

3.3.2 Artifactual Communication in the Extended Memory Hierarchy

Data accesses that are not satisfied in the local (on-node) portion of the extended memory hierarchy generate communication. Inherent communication can be seen as part of this: The data moves from one processor, through the memory hierarchy in some manner, to another processor, regardless of whether it does this through explicit messages or reads and writes. However, the amount of communication that occurs in an execution of the program is usually greater than the inherent interprocess communication. The additional communication is an artifact of how the program is actually implemented and how it interacts with the machine's extended memory hierarchy. There are many sources of this *artifactual* communication:

Poor allocation of data. Data accessed by one node may happen to be allocated in the local memory of another. Accesses to remote data involve communication, even if the data is not updated by other nodes. Such transfer can be eliminated by a better assignment or better distribution of data, or reduced by replicating the data when it is accessed.

Unnecessary data in a transfer. More data than needed may be communicated in a transfer. For example, a receiver may not use all the data in a message: It may have been easier to send extra data conservatively than determine exactly what to send. Similarly, if data is transferred implicitly in units larger than a word, e.g. cache blocks, part of the block may not be used by the requester. This artifactual communication can be eliminated with smaller transfers.

Unnecessary transfers due to system granularities. In cache-coherent machines, data may be kept coherent at a granularity larger than a single word, which may lead to extra communication to keep data coherent as we shall see in later chapters.

Redundant communication of data. Data may be communicated multiple times, for example, every time the value changes, but only the last value may actually be used. On the other hand, data may be communicated to a process that already has the latest values, again because it was too difficult to determine this.

Finite replication capacity. The capacity for replication on a node is finite—whether it be in the cache or the main memory—so data that has been already communicated to a process may be replaced from its local memory system and hence need to be transferred again even if it has not since been modified.

In contrast, inherent communication is that which would occur with unlimited capacity for replication, transfers as small as necessary, and perfect knowledge of what logically shared data has been updated. We will understand some of the sources of artifactual communication better when we get deeper into architecture. Let us look a little further at the last source of artifactual communication—finite replication capacity—which has particularly far-reaching consequences.

Artifactual Communication and Replication: The Working Set Perspective

The relationship between finite replication capacity and artifactual communication is quite fundamental in parallel systems, just like the relationship between cache size and memory traffic in uniprocessors. It is almost inappropriate to speak of the amount of communication without reference to replication capacity. The extended memory hierarchy perspective is useful in viewing this

relationship. We may view our generic multiprocessor as a hierarchy with three levels: Local cache is inexpensive to access, local memory is more expensive, and any remote memory is much more expensive. We can think of any level as a cache, whether it is actually managed like a hardware cache or by software. Then, we can then classify the "misses" at any level, which generate traffic to the next level, just as we do for uniprocessors. A fraction of the traffic at any level is *cold-start*, resulting from the first time data is accessed by the processor. This component, also called *compulsory* traffic in uniprocessors, is independent of cache size. Such cold start misses are a concern in performing cache simulations, but diminish in importance as programs run longer. Then there is traffic due to *capacity* misses, which clearly decrease with increases in cache size. A third fraction of traffic may be *conflict* misses, which are reduced by greater associativity in the replication store, a greater number of blocks, or changing the data access pattern. These three types of misses or traffic are called the three C's in uniprocessor architecture (cold start or compulsory, capacity, conflict). The new form of traffic in multiprocessors is a fourth C, a *communication* miss, caused by the inherent communication between processors or some of the sources of artifactual communication discussed above. Like cold start misses, communication misses do not diminish with cache size. Each of these components of traffic may be helped or hurt by large granularities of data transfer depending on spatial locality.

If we were to determine the traffic resulting from each type of miss for a parallel program as we increased the replication capacity (or cache size), we could expect to obtain a curve such as shown in Figure 3-6. The curve has a small number of knees or points of inflection. These knees correspond to the *working sets* of the algorithm relevant to that level of the hierarchy.[1] For the first-level cache, they are the working sets of the algorithm itself; for others they depend on how references have been filtered by other levels of the hierarchy and on how the levels are managed. We speak of this curve for a first-level cache (assumed fully associative with a one-word block size) as the working set curve for the algorithm.

Traffic resulting from any of these types of misses may cause communication across the machine's interconnection network, for example if the backing storage happens to be in a remote node. Similarly, any type of miss may contribute to local traffic and data access cost. Thus, we might expect that many of the techniques used to reduce artifactual communication are similar to those used to exploit locality in uniprocessors, with some additional ones. Inherent communication misses almost always generate actual communication in the machine (though sometimes they may not if the data needed happens to have become local in the meanwhile, as we shall see), and can only be reduced by changing the sharing patterns in the algorithm. In addition, we are strongly motivated to reduce the artifactual communication that arises either due to transfer size or due to limited replication capacity, which we can do by exploiting spatial and temporal locality in a process's data accesses in the extended hierarchy. Changing the orchestration and assignment can dramatically change locality characteristics, including the shape of the working set curve.

For a given amount of communication, its cost as seen by the processor is also affected by how the communication is structured. By structure we mean whether messages are large or small, how

---

1. The working set model of program behavior [Den68] is based on the temporal locality exhibited by the data referencing patterns of programs. Under this model, a program—or a process in a parallel program—has a set of data that it reuses substantially for a period of time, before moving on to other data. The shifts between one set of data and another may be abrupt or gradual. In either case, there is at most times a "working set" of data that a processor should be able to maintain in a fast level of the memory hierarchy, to use that level effectively.
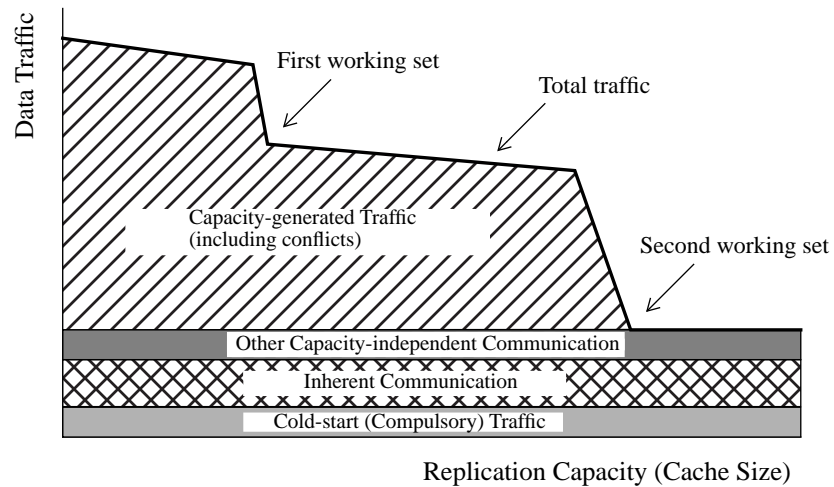
Figure 3-6 The data traffic between a cache (replication store) and the rest of the system, and its components as a function of cache size.

The points of inflection in the total traffic curve indicate the working sets of the program.

bursty the communication is, whether communication cost can be overlapped with other computation or communication—all of which are addressed in the orchestration step—and how well the communication patterns match the topology of the interconnection network, which is addressed in the mapping step. Reducing the amount of communication—inherent or artifactual—is important because it reduces the demand placed on both the system and the programmer to reduce cost. Having understood the machine as an extended hierarchy and the major issues this raises, let us see how to address these architecture-related performance issues at the next level in software; that is, how to program for performance once partitioning issues are resolved.

3.4 Orchestration for Performance

We begin by discussing how we might exploit temporal and spatial locality to reduce the amount of artifactual communication, and then move on to structuring communication—inherent or artifactual—to reduce cost.

3.4.1 Reducing Artifactual Communication

In the message passing model both communication and replication are explicit, so even artifactual communication is explicitly coded in program messages. In a shared address space, artifactual communication is more interesting architecturally since it occurs transparently due to interactions between the program and the machine organization; in particular, the finite cache size and the granularities at which data are allocated, communicated and kept coherent. We therefore use a shared address space to illustrate issues in exploiting locality, both to improve node performance and to reduce artifactual communication.

Exploiting Temporal Locality

A program is said to exhibit temporal locality if tends to access the same memory location repeatedly in a short time-frame. Given a memory hierarchy, the goal in exploiting temporal locality is to structure an algorithm so that its working sets map well to the sizes of the different levels of the hierarchy. For a programmer, this typically means keeping working sets small without losing performance for other reasons. Working sets can be reduced by several techniques. One is the same technique that reduces inherent communication—assigning tasks that tend to access the same data to the same process—which further illustrates the relationship between communication and locality. Once assignment is done, a process's assigned computation can be organized so that tasks that access the same data are scheduled close to one another in time, and so that we reuse a set of data as much as possible before moving on to other data, rather than moving on and coming back.

When multiple data structures are accessed in the same phase of a computation, we must decide which are the most important candidates for exploiting temporal locality. Since communication is more expensive than local access, we might prefer to exploit temporal locality on nonlocal rather than local data. Consider a database application in which a process wants to compare all its records of a certain type with all the records of other processes. There are two choices here: (i) for each of its own records, the process can sweep through all other (nonlocal) records and compare, and (ii) for each nonlocal record, the process can sweep through its own records and compare. The latter exploits temporal locality on nonlocal data, and is therefore likely to yield better overall performance.

Example 3-1   To what extent is temporal locality exploited in the equation solver kernel. How might the temporal locality be increased?

Answer   The equation solver kernel traverses only a single data structure. A typical grid element in the interior of a process's partition is accessed at least five times by that process during each sweep: at least once to compute its own new value, and once each to compute the new values of its four nearest neighbors. If a process sweeps through its partition of the grid in row-major order (i.e. row by row and left to right within each row, see Figure 3-7(a)) then reuse of $A[i,j]$ is guaranteed across the updates of the three elements in the same row that touch it: $A[i,j-1]$, $A[i,j]$ and $A[i,j+1]$. However, between the times that the new values for $A[i,j]$ and $A[i+1,j]$ are computed, three whole subrows of elements in that process's partition are accessed by that process. If the three subrows don't fit together in the cache, then $A[i,j]$ will no longer be in the cache when it is accessed again to compute $A[i+1,j]$. If the backing store for these data in the cache is nonlocal, artifactual communication will result. The problem can be fixed by changing the order in which elements are computed, as shown in Figure 3-7(b). Essentially, a process proceeds left-to-right not for the length of a whole subrow of its partition, but only a certain length B, before it moves on to the corresponding portion of the next subrow. It performs its sweep in sub-sweeps over B-by-B blocks of its partition. The

(a) Unblocked access pattern in a sweep   (b) Blocked access pattern with B=4
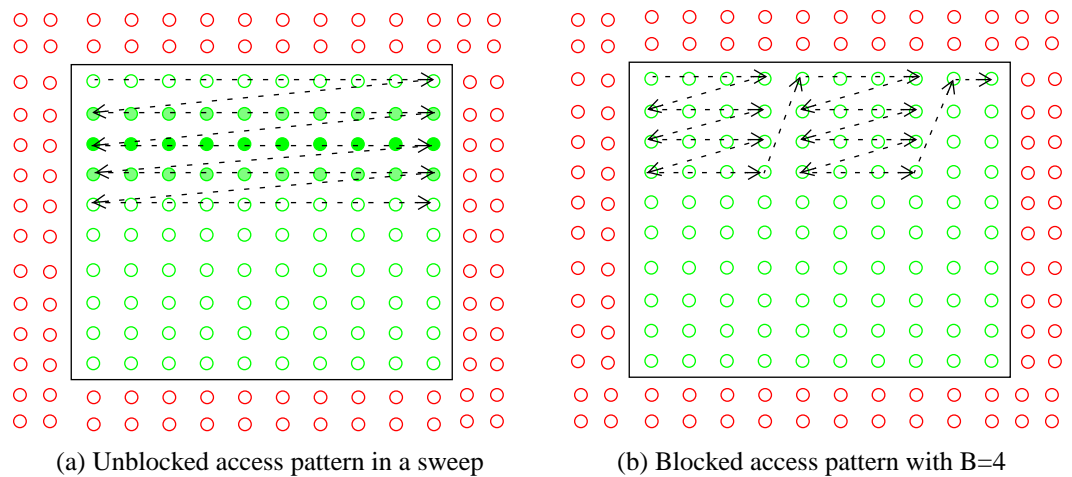
Figure 3-7 Blocking to exploit temporal locality in the equation solver kernel.

The figures show the access patterns for a process traversing its partition during a sweep, with the arrow-headed lines showing the order in which grid points are updated. Updating the subrow of bold elements requires accessing that subrow as well as the two sub-rows of shaded elements. Updating the first element of the next (shaded) subrow requires accessing the first element of the bold sub-row again, but these three whole subrows have been accessed since the last time that first bold element was accessed.

block size B is chosen so that at least three B-length rows of a partition fit in the cache.

This technique of structuring computation so that it accesses a subset of data that fits in a level of the hierarchy, uses those data as much as possible, and then moves on to the next such set of data, is called *blocking*. In the particular example above, the reduction in miss rate due to blocking is only a small constant factor (about a factor of two). The reduction is only seen when a subrow of a process's partition of a grid itself does not fit in the cache, so blocking is not always useful. However, blocking is used very often in linear algebra computations like matrix multiplication or matrix factorization, where $O(n^{k+1})$ computation is performed on a data set of size $O(n^k)$, so each data item is accessed $O(n)$ times. Using blocking effectively with B-by-B blocks in these cases can reduce the miss rate by a factor of B, as we shall see in Exercise 3.5, which is particularly important since much of the data accessed is nonlocal. Not surprisingly, many of the same types of restructuring are used to improve temporal locality in a sequential program as well; for example, blocking is critical for high performance in sequential matrix computations. Techniques for temporal locality can be used at any level of the hierarchy where data are replicated—including main memory—and for both explicit or implicit replication.

Since temporal locality affects replication, the locality and referencing patterns of applications have important implications for determining which programming model and communication abstraction a system should support. We shall return to this issue in Section 3.7. The sizes and scaling of working sets have obvious implications for the amounts of replication capacity needed at different levels of the memory hierarchy, and the number of levels that make sense in this hier-archy. In a shared address space, together with their compositions (whether they hold local or remote data or both), working set sizes also help determine whether it is useful to replicate com-municated data in main memory as well or simply rely on caches, and if so how this should be done. In message passing, they help us determine what data to replicate and how to manage the

replication so as not to fill memory with replicated data. Of course it is not only the working sets of individual applications that matter, but those of the entire workloads and the operating system that run on the machine. For hardware caches, the size of cache needed to hold a working set depends on its organization (associativity and block size) as well.

Exploiting Spatial Locality

A level of the extended memory hierarchy exchanges data with the next level at a certain *granularity* or transfer size. This granularity may be fixed (for example, a cache block or a page of main memory) or flexible (for example, explicit user-controlled messages or user-defined objects). It usually becomes larger as we go further away from the processor, since the latency and fixed startup overhead of each transfer becomes greater and should be amortized over a larger amount of data. To exploit a large *granularity of communication or data transfer*, we should organize our code and data structures to exploit spatial locality.[1] Not doing so can lead to artifactual communication if the transfer is to or from a remote node and is implicit (at some fixed granularity), or to more costly communication even if the transfer is explicit (since either smaller messages may have to be sent or the data may have to be made contiguous before they are sent). As in uniprocessors, poor spatial locality can also lead to a high frequency of TLB misses.

In a shared address space, in addition to the granularity of communication, artifactual communication can also be generated by mismatches of spatial locality with two other important granularities. One is the *granularity of allocation*, which is the granularity at which data are allocated in the local memory or replication store (e.g. a page in main memory, or a cache block in the cache). Given a set of data structures, this determines the granularity at which the data can be distributed among physical main memories; that is, we cannot allocate a part of a page in one node's memory and another part of the page in another node's memory. For example, if two words that are mostly accessed by two different processors fall on the same page, then unless data are replicated in multiple main memories that page may be allocated in only one processor's local memory; capacity or conflict cache misses to that word by the other processor will generate communication. The other important granularity is the *granularity of coherence*, which we will discuss in Chapter 5. We will see that unrelated words that happen to fall on the same unit of coherence in a coherent shared address space can also cause artifactual communication.

The techniques used for all these aspects of spatial locality in a shared address space are similar to those used on a uniprocessor, with one new aspect: We should try to keep the data accessed by a given processor close together (contiguous) in the address space, and unrelated data accessed by different processors apart. Spatial locality issues in a shared address space are best examined in the context of particular architectural styles, and we shall do so in Chapters 5 and 8. Here, for illustration, we look at one example: how data may be restructured to interact better with the granularity of allocation in the equation solver kernel.

---

1. The principle of spatial locality states that if a given memory location is referenced now, then it is likely that memory locations close to it will be referenced in the near future.It should be clear that what is called spatial locality at the granularity of individual words can also be viewed as temporal locality at the granularity of cache blocks; that is, if a cache block is accessed now, then it (another datum on it) is likely to be accessed in the near future.

Example 3-2   Consider a shared address space system in which main memory is physically distributed among the nodes, and in which the granularity of allocation in main memory is a page (4 KB, say). Now consider the grid used in the equation solver kernel. What is the problem created by the granularity of allocation, and how might it be addressed?

Answer   The natural data structure with which to represent a two-dimensional grid in a shared address space, as in a sequential program, is a two-dimensional array. In a typical programming language, a two-dimensional array data structure is allocated in either a "row-major" or "column-major" way.[1] The gray arrows in Figure 3-8(a) show the contiguity of virtual addresses in a row-major allocation, which is the one we assume. While a two-dimensional shared array has the programming advantage

Contiguity in memory layout

Page straddles partition boundaries; difficult to distribute memory well

Cache block straddles partition boundary

Page does not straddle partition boundaries

Cache block is within a partition

(a) Two-dimensional array   (a) Four-dimensional array
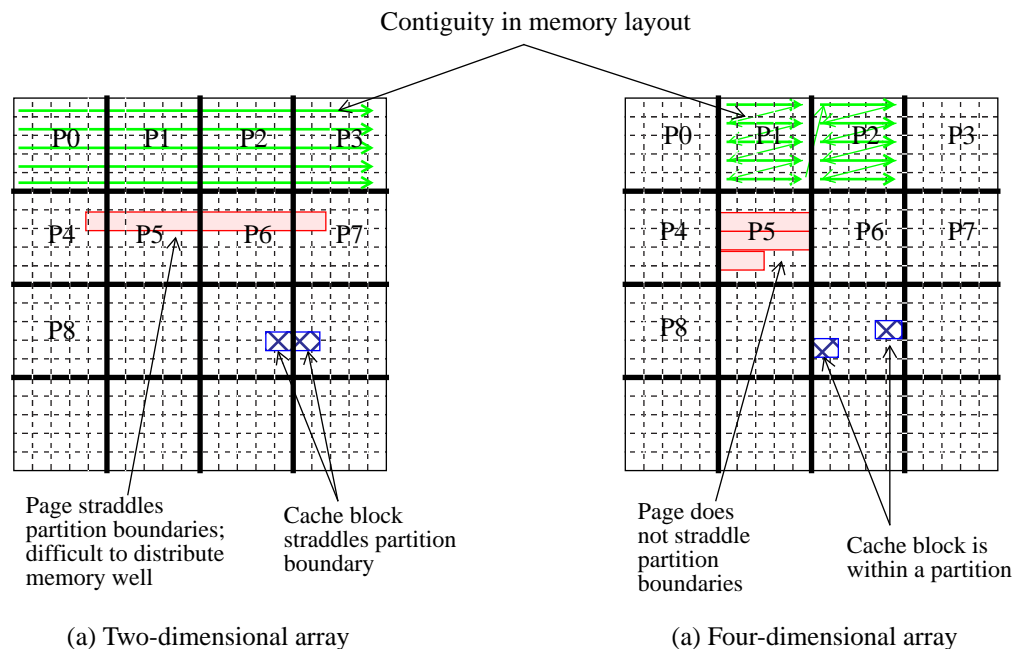
Figure 3-8 Two-dimensional and Four-dimensional arrays used to represent a two-dimensional grid in a shared address space.

of being the same data structure used in a sequential program, it interacts poorly

---

1. Consider the array as being a two-dimensional grid, with the first dimension specifying the row number in the grid and the second dimension the column number. Row-major allocation means that all elements in the first row are contiguous in the virtual address space, followed by all the elements in the second row, etc. The C programming language, which we assume here, is a row-major language (FORTRAN, for example, is column-major).

with the granularity of allocation on a machine with physically distributed memory (and with other granularities as well, as we shall see in Chapter 5).

Consider the partition of processor *P5* in Figure 3-8(a). An important working set for the processor is its entire partition, which it streams through in every sweep and reuses across sweeps. If its partition does not fit in its local cache hierarchy, we would like it to be allocated in local memory so that the misses can be satisfied locally. The problem is that consecutive subrows of this partition are not contiguous with one another in the address space, but are separated by the length of an entire row of the grid (which contains subrows of other partitions). This makes it impossible to distribute data appropriately across main memories if a subrow of a partition is either smaller than a page, or not a multiple of the page size, or not well aligned to page boundaries. Subrows from two (or more) adjacent partitions will fall on the same page, which at best will be allocated in the local memory of one of those processors. If a processor's partition does not fit in its cache, or it incurs conflict misses, it may have to communicate every time it accesses a grid element in its own partition that happens to be allocated nonlocally.

The solution in this case is to use a higher-dimensional array to represent the two-dimensional grid. The most common example is a four-dimensional array, in which case the processes are arranged conceptually in a two-dimensional grid of partitions as seen in Figure 3-8. The first two indices specify the partition or process being referred to, and the last two represent the subrow and subcolumn numbers within that partition. For example, if the size of the entire grid is 1024-by-1024 elements, and there are 16 processors, then each partition will be a subgrid of size $\frac{1024}{\sqrt{16}} - \text{by} - \frac{1024}{\sqrt{16}}$ or 256-by-256 elements. In the four-dimensional representation, the array will be of size 4-by-4-by-256-by-256 elements. The key property of these higher-dimensional representations is that each processor's 256-by-256 element partition is now contiguous in the address space (see the contiguity in the virtual memory layout in Figure 3-8(b)). The data distribution problem can now occur only at the end points of entire partitions, rather than of each subrow, and does not occur at all if the data structure is aligned to a page boundary. However, it is substantially more complicated to write code using the higher-dimensional arrays, particularly for array indexing of neighboring process's partitions in the case of the nearest-neighbor computation.

More complex applications and data structures illustrate more significant tradeoffs in data structure design for spatial locality, as we shall see in later chapters.

The spatial locality in processes' access patterns, and how they scale with the problem size and number of processors, affects the desirable sizes for various granularities in a shared address space: allocation, transfer and coherence. It also affects the importance of providing support tailored toward small versus large messages in message passing systems. Amortizing hardware and transfer cost pushes us toward large granularities, but too large granularities can cause performance problems, many of them specific to multiprocessors. Finally, since conflict misses can generate artifactual communication when the backing store is nonlocal, multiprocessors push us toward higher associativity for caches. There are many cost, performance and programmability tradeoffs concerning support for data locality in multiprocessors—as we shall see in subsequent chapters—and our choices are best guided by the behavior of applications.

Finally, there are often interesting tradeoffs among algorithmic goals such as minimizing inherent communication, implementation issues, and architectural interactions that generate artifactual communication, suggesting that careful examination of tradeoffs is needed to obtain the best performance on a given architecture. Let us illustrate using the equation solver kernel.
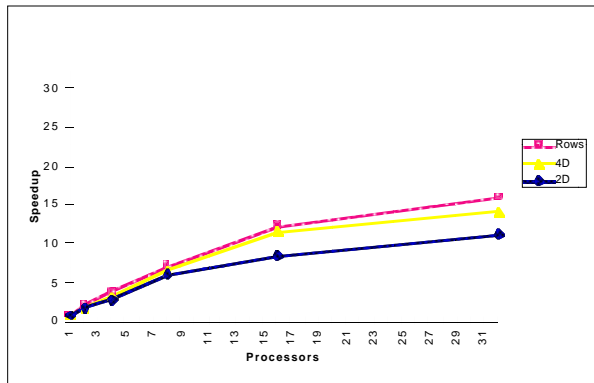
(a) Ocean with 514-by-514 grids
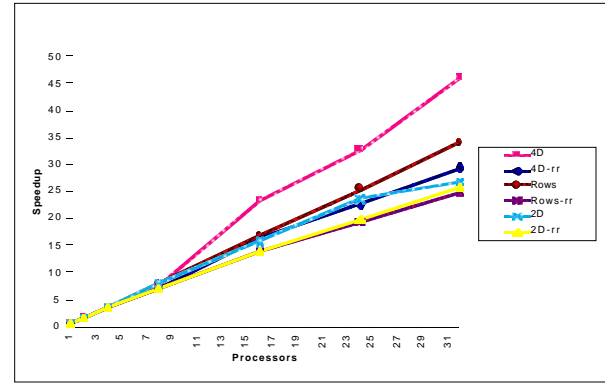
(b) Equation solver kernel with 12K-by-12K grid

Figure 3-9 The impact of data structuring and spatial data locality on performance.

All measurements are on the SGI Origin2000. 2D and 4D imply two- and four-dimensional data structures, respectively, with subblock assignment. Rows uses the two-dimensional array with strip assignment into chunks of rows. In the right graph, the postfix rr means that pages of data are distributed round-robin among physical memories. Without rr, it means that pages are placed in the local memory of the processor to which their data is assigned, as far as possible. We see from (a) that the rowwise strip assignment outperforms the 2D strip assignment due to spatial locality interactions with long cache blocks (128 bytes on the Origin2000), and even a little better than the 4D array block assignment due to poor spatial locality in the latter in accessing border elements at column-oriented partition boundaries. The right graph shows that in all partitioning schemes proper data distribution in main memory is important to performance, though least successful for the 2D array subblock partitions. In the best case, we see superlinear speedups once there are enough processors that the size of a processor's partition of the grid (its important working set) fits into its cache.

Example 3-3   Given the performance issues discussed so far, should we choose to partition the equation solver kernel into square-like blocks or into contiguous strips of rows?

Answer   If we only consider inherent communication, we already know that a block domain decomposition is better than partitioning into contiguous strips of rows (see Figure 3-5 on page 142). However, a strip decomposition has the advantage that it keeps a partition wholly contiguous in the address space even with the simpler, two-dimensional array representation. Hence, it does not suffer problems related to the interactions of spatial locality with machine granularities, such as the granularity of allocation as discussed above. This particular interaction in the block case can of course be solved by using a higher-dimensional array representation. However, a more difficult interaction to solve is with the granularity of communication. In a subblock assignment, consider a neighbor element from another partition at a column-oriented partition boundary. If the granularity of communication is large, then when a process references this element from its neighbor's partition it will fetch not only that element but also a number of other elements that are on the same unit of communication. These other elements are not neighbors of the fetching process's partition, regardless of whether a two-dimensional or four-dimensional

representation is used, so they are useless and waste communication bandwidth (see Figure 3-10). With a partitioning into strips of rows, a referenced element still
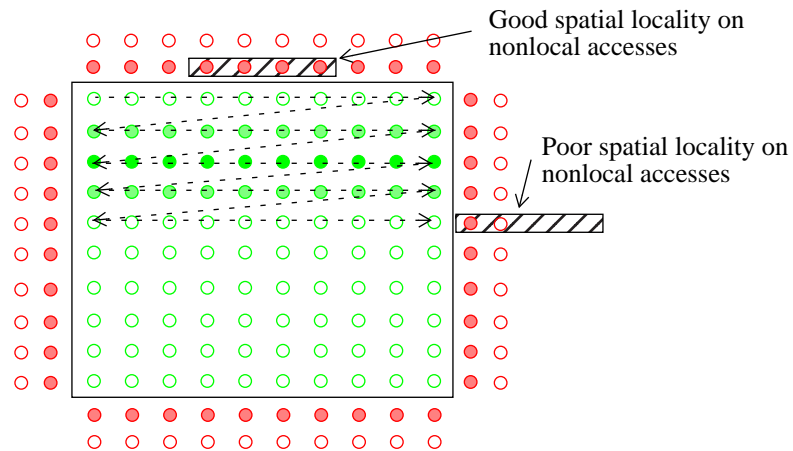
Figure 3-10 Spatial locality in accesses to nonlocal data in the equation solver kernel.

The shaded points are the nonlocal points that the processor owning the partition accesses. The hatched rectangles are cache blocks, showing good spatial locality along the row boundary but poor along the column.

causes other elements from its row to be fetched, but now these elements are indeed neighbors of the fetching process's partition. They are useful, and in fact the large granularity of communication results in a valuable prefetching effect. Overall, there are many combinations of application and machine parameters for which the performance losses in block partitioning owing to artifactual communication will dominate the performance benefits from reduced inherent communication, so that a strip partitioning will perform better than a block partitioning on a real system. We might imagine that it should most often perform better when a two-dimensional array is used in the block case, but it may also do so in some cases when a four-dimensional array is used (there is not motivation to use a four-dimensional array with a strip partitioning). Thus, artifactual communication may cause us to go back and revise our partitioning method from block to strip. Figure 3-9(a) illustrates this effect for the Ocean application, and Figure 3-9(b) uses the equation solver kernel with a larger grid size to also illustrate the impact of data placement. Note that a strip decomposition into columns rather than rows will yield the worst of both worlds when data are laid out in memory in row-major order.

3.4.2 Structuring Communication to Reduce Cost

Whether communication is inherent or artifactual, how much a given communication-to-computation ratio contributes to execution time is determined by how the communication is organized or structured into messages. A small communication-to-computation ratio may have a much greater impact on execution time than a large ratio if the structure of the latter interacts much better with the system. This is an important issue in obtaining good performance from a real machine, and is the last major performance issue we examine. Let us begin by seeing what the structure of communication means.

In Chapter 1, we introduced a model for the cost of communication as seen by a processor given a frequency of program initiated communication operations or messages (initiated explicit messages or loads and stores). Combining equations 1.5 and 1.6, that model for the cost C is:

$$C = freq \times \left(Overhead + Delay + \frac{Length}{Bandwidth} + Contention - Overlap \right)$$

or

$$C = f \times \left(o + l + \frac{n_c/m}{B} + t_c - overlap \right) \qquad \textbf{(EQ 3.4)}$$

where

f is the frequency of communication messages in the program.

n_c is the total amount of data communicated by the program.

m is the number of message, so n_c/m is the average length of a message.

o is the combined overhead of handling the initiation and reception of a message at the sending and receiving processors, assuming no contention with other activities.

l is the delay for the first bit of the message to reach the destination processor or memory, which includes the delay through the assists and network interfaces as well as the network latency or transit latency in the network fabric itself (a more detailed model might separate the two out). It assumes no contention.

B is the point-to-point bandwidth of communication afforded for the transfer by the communication path, excluding the processor overhead; i.e. the rate at which the rest of the message data arrives at the destination after the first bit, assuming no contention. It is the inverse of the overall occupancy discussed in Chapter 1. It may be limited by the network links, the network interface or the communication assist.

t_c is the time induced by contention for resources with other activities.

overlap is the amount of the communication cost that can be overlapped with computation or other communication, i.e that is not in the critical path of a processor's execution.

This expression for communication cost can be substituted into the speedup equation we developed earlier (Equation 3.1 on page 144), to yield our final expression for speedup. The portion of the cost expression inside the parentheses is our cost model for a single data one-way message. It assumes the "cut-through" or pipelined rather than store-and-forward transmission of modern multiprocessor networks. If messages are round-trip, we must make the appropriate adjustments. in addition to reducing communication volume *(n_c)* our goals in structuring communication may include (i) reducing communication overhead (*m*o*), (ii) reducing latency (*m*l*), (iii) reducing contention (*m*t_c*), and (iv) overlapping communication with computation or other communication to hide its latency. Let us discuss programming techniques for addressing each of the issues.

Reducing Overhead

Since the overhead *o* associated with initiating or processing a message is usually fixed by hardware or system software, the way to reduce communication overhead is to make messages fewer in number and hence larger, i.e. to reduce *t*he message frequency[1] Explicitly initiated communication allows greater flexibility in specifying the sizes of messages; see the SEND primitive described in Section 2.4.6 in the previous chapter. On the other hand, implicit communication

through loads and stores does not afford the program direct control, and the system must take responsibility for coalescing the loads and stores into larger messages if necessary.

Making messages larger is easy in applications that have regular data access and communication patterns. For example, in the message passing equation solver example partitioned into rows we sent an entire row of data in a single message. But it can be difficult in applications that have irregular and unpredictable communication patterns, such as Barnes-Hut or Raytrace. As we shall see in Section 3.7, it may require changes to the parallel algorithm, and extra work to determine which data to coalesce, resulting in a tradeoff between the cost of this computation and the savings in overhead. Some computation may be needed to determine what data should be sent, and the data may have to be gathered and packed into a message at the sender and unpacked and scattered into appropriate memory locations at the receiver.

Reducing Delay

Delay through the assist and network interface can be reduced by optimizing those components. Consider the network transit delay or the delay through the network itself. In the absence of contention and with our pipelined network assumption, the transit delay l of a bit through the network itself can be expressed as $h*t_h$, where h is the number of "hops" between adjacent network nodes that the message traverses, t_h is the delay or latency for a single bit of data to traverse a single network hop, including the link and the router or switch. Like o above, t_h is determined by the system, and the program must focus on reducing the f and h components of the $f*h*t_h$ delay cost. (In store-and-forward networks, t_h would be the time for the entire message to traverse a hop, not just a single bit.)

The number of hops h can be reduced by *mapping* processes to processors so that the topology of interprocess communication in the application exploits locality in the physical topology of the network. How well this can be done in general depends on application and on the structure and richness of the network topology; for example, the nearest-neighbor equation solver kernel (and the Ocean application) would map very well onto a mesh-connected multiprocessor but not onto a unidirectional ring topology. Our other example applications are more irregular in their communication patterns. We shall see several different topologies used in real machines and discuss their tradeoffs in Chapter 10.

There has been a lot of research in mapping algorithms to network topologies, since it was thought that as the number of processors p became large poor mappings would cause the latency due to the $h*t_h$ term to dominate the cost of messages. How important topology actually is in practice depends on several factors: (i) how large the t_h term is relative to the overhead o of getting a message into and out of the network, (ii) the number of processing nodes on the machine, which determines the maximum number of hops h for a given topology, and (iii) whether the machine is used to run a single application at a time in "batch" mode or is multiprogrammed among applications. It turns out that network topology is not considered as important on modern machines as it once was, because of the characteristics of the machines along all three axes: overhead dominates hop latency (especially in machines that do not provide hardware support for a shared address space), the number of nodes is usually not very large, and the machines are often

---

1. Some explicit message passing systems provide different types of messages with different costs and functionalities among which a program can choose.

used as general-purpose, multiprogrammed servers. Topology-oriented design might not be very useful in multiprogrammed systems, since the operating system controls resource allocation dynamically and might transparently change the mapping of processes to processors at runtime. For these reasons, the mapping step of parallelization receives considerably less attention than decomposition, assignment and orchestration. However, this may change again as technology and machine architecture evolve. Let us now look at contention, the third major cost issue in communication.

Reducing Contention

The communication systems of multiprocessors consist of many resources, including network links and switches, communication controllers, memory systems and network interfaces. All of these resources have a nonzero *occupancy*, or time for which they are occupied servicing a given transaction. Another way of saying this is that they have finite bandwidth (or rate, which is the reciprocal of occupancy) for servicing transactions. If several messages contend for a resource, some of them will have to wait while others are serviced, thus increasing message latency and reducing the bandwidth available to any single message. Resource occupancy contributes to message cost even when there is no contention, since the time taken to pass through resource is part of the delay or overhead, but it can also cause contention.

Contention is a particularly insidious performance problem, for several reasons. First, it is easy to ignore when writing a parallel program, particularly if it is caused by artifactual communication. Second, its effect on performance can be dramatic. If p processors contend for a resource of occupancy x, the first to obtain the resource incurs a latency of x due to that resource, while the last incurs at least $p*x$. In addition to large stall times for processors, these differences in wait time across processors can also lead to large load imbalances and synchronization wait times. Thus, the contention caused by the occupancy of a resource can be much more dangerous than just the latency it contributes in uncontended cases. The third reason is that contention for one resource can hold up other resources, thus stalling transactions that don't even need the resource that is the source of the contention. This is similar to how contention for a single-lane exit off a multi-lane highway causes congestion on the entire stretch of highway. The resulting congestion also affects cars that don't need that exit but want to keep going on the highway, since they may be stuck behind cars that do need that exit. The resulting backup of cars covers up other unrelated resources (previous exits), making them inaccessible and ultimately clogging up the highway. Bad cases of contention can quickly saturate the entire communication architecture. The final reason is related: The cause of contention is particularly difficult to identify, since the effects might be felt at very different points in the program than the original cause, particularly communication is implicit.

Like bandwidth, contention in a network can also be viewed as being of two types: at links or switches within the network, called *network contention*, and at the end-points or processing nodes, called *end-point contention*. Network contention, like latency, can be reduced by mapping processes and scheduling the communication appropriately in the network topology. End-point contention occurs when many processors need to communicate with the same processing node at the same time (or when communication transactions interfere with local memory references). When this contention becomes severe, we call that processing node or resource a *hot spot*. Let us examine a simple example of how a hot spot may be formed, and how it might be alleviated in software.

Recall the case of processes want to accumulate their partial sums into a global sum, as in our equation solver kernel. The resulting contention for the global sum can be reduced by using tree-structured communication rather than having all processes send their updates to the owning node directly. Figure 3-11 shows the structure of such many-to-one communication using a binary fan-in tree. The nodes of this tree, which is often called a software combining tree, are the participating processes. A leaf process sends its update up to its parent, which combines its children's updates with its own and sends the combined update up to its parent, and so on till the updates reach the root (the process that holds the global sum) in $\log_2 p$ steps. A similar fan-out tree can be used to send data from one to many processes. Similar tree-based approaches are used to design
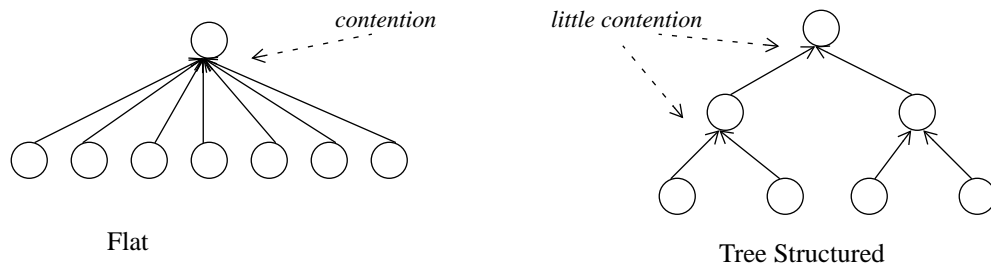
Figure 3-11 Two ways of structuring many-to-one communication: flat, and tree-structured in a binary fan-in tree.

Note that the destination processor may received up to *p-1* messages at a time in the flat case, while no processor is the destination of more than two messages in the binary tree.

scalable synchronization primitives like locks and barriers that often experience a lot of contention, as we will see in later chapters, as well as library routines for other communication patterns.

In general, two principles for alleviating contention are to avoid having too many processes communicate with the same process at the same time, and to stagger messages to the same destination so as not to overwhelm the destination or the resources along the way. Contention is often caused when communication is bursty (i.e. the program spends some time not communicating much and then suddenly goes through a burst of communication), and staggering reduces burstiness. However, this must be traded off with making messages large, which tends to increase burstiness. Finally, having discussed overhead, latency, and contention, let us look at the last of the communication cost issues.

Overlapping Communication with Computation or Other Communication

Despite efforts to reduce overhead and delay, the technology trends discussed in Chapter 1 suggest that the end-to-end the communication cost as seen by a processor is likely to remain very large in processor cycles. Already, it is in the hundreds of processor cycles even on machines that provide full hardware support for a shared address space and use high-speed networks, and is at least an order of magnitude higher on current message passing machines due to the higher overhead term *o*. If the processor were to remain idle (stalled) while incurring this cost for every word of data communicated, only programs with an extremely low ratio of communication to computation would yield effective parallel performance. Programs that communicate a lot must therefore find ways to hide the cost of communication from the process's critical path by overlapping

it with computation or other communication as much as possible, and systems must provide the necessary support.

Techniques to hide communication cost come in different, often complementary flavors, and we shall spend a whole chapter on them. One approach is simply to make messages larger, thus incurring the latency of the first word but hiding that of subsequent words through pipelined transfer of the large message. Another, which we can call precommunication, is to initiate the communication much before the data are actually needed, so that by the time the data are needed they are likely to have already arrived. A third is to perform the communication where it naturally belongs in the program, but hide its cost by finding something else for the processor to do from later in the same process (computation or other communication) while the communication is in progress. A fourth, called multithreading is to switch to a different thread or process when one encounters a communication event. While the specific techniques and mechanisms depend on the communication abstraction and the approach taken, they all fundamentally require the program to have extra concurrency (also called *slackness*) beyond the number of processors used, so that independent work can be found to overlap with the communication.

Much of the focus in parallel architecture has in fact been on reducing communication cost as seen by the processor: reducing communication overhead and latency, increasing bandwidth, and providing mechanisms to alleviate contention and overlap communication with computation or other communication. Many of the later chapters will therefore devote a lot of attention to covering these issues—including the design of node to network interfaces and communication protocols and controllers that minimize both software and hardware overhead (Chapter 7, 8 and 9), the design of network topologies, primitive operations and routing strategies that are well-suited to the communication patterns of applications (Chapter 10), and the design of mechanisms to hide communication cost from the processor (Chapter 11). The architectural methods are usually expensive, so it is important that they can be used effectively by real programs and that their performance benefits justify their costs.

3.5 Performance Factors from the Processors' Perspective

To understand the impact of different performance factors in a parallel program, it is useful to look from an individual processor's viewpoint at the different components of time spent executing the program; i.e. how much time the processor spends in different activities as it executes instructions and accesses data in the extended memory hierarchy. These different components of time can be related quite directly to the software performance issues studied in this chapter, helping us relate software techniques to hardware performance. This view also helps us understand what a parallel execution looks like as a workload presented to the architecture, and will be useful when we discuss workload-driven architectural evaluation in the next chapter.

In Equation 3.3, we described the time spent executing a sequential program on a uniprocessor as the sum of the time actually executing instructions (*busy*) and the time stalled on the memory system (*data-local*), where the latter is a "non-ideal" factor that reduces performance. Figure 3-12(a) shows a profile of a hypothetical sequential program. In this case, about 80% of the execution time is spent performing instructions, which can be reduced only by improving the algorithm or the processor. The other 20% is spent stalled on the memory system, which can be improved by improving locality or the memory system.

In multiprocessors we can take a similar view, though there are more such non-ideal factors. This view cuts across programming models: for example, being stalled waiting for a receive to complete is really very much like being stalled waiting for a remote read to complete or a synchronization event to occur. If the same program is parallelized and run on a four-processor machine, the execution time profile of the four processors might look like that in Figure 3-12(b). The figure

(a) Sequential   (b) Parallel with four processors
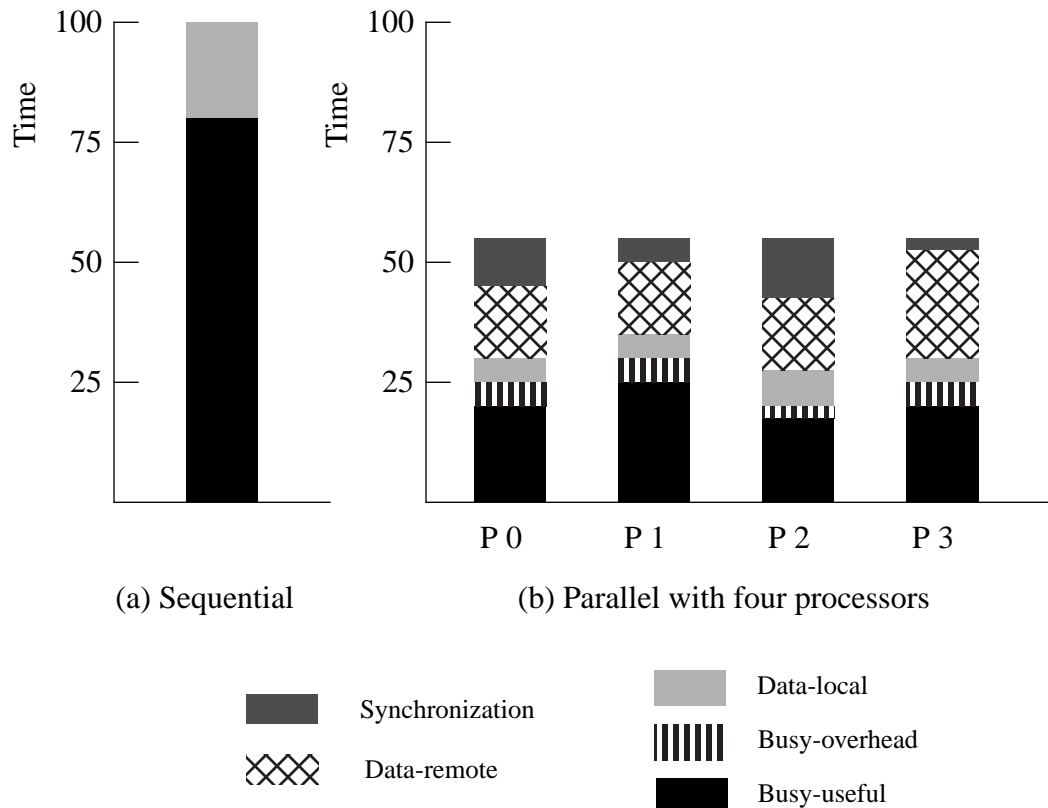
Figure 3-12 Components of execution time from the perspective of an individual processor.

assumes a global synchronization point at the end of the program, so that all processes terminate at the same time. Note that the parallel execution time (55 sec) is greater than one-fourth of the sequential execution time (100 sec); that is, we have obtained a speedup of only $\frac{100}{55}$ or 1.8 instead of the four-fold speedup for which we may have hoped. Why this is the case, and what specific software or programming factors contribute to it can be determined by examining the components of parallel execution time from the perspective of an individual processor. These are:

Busy-useful: this is the time that the processor spends executing instructions that would have been executed in the sequential program as well. Assuming a deterministic parallel program[1] that is derived directly from the sequential algorithm, the sum of the busy-useful times for all processors is equal to the busy-useful time for the sequential execution. (This is true at least for processors that execute a single instruction per cycle; attributing cycles of execution time to busy-useful or other categories is more complex when processors issue multiple instruc-

tions per cycle, since different cycles may execute different numbers of instructions and a given stall cycle may be attributable to different causes. We will discuss this further in Chapter 11 when we present execution time breakdowns for superscalar processors.)

Busy-overhead: the time that the processor spends executing instructions that are not needed in the sequential program, but only in the parallel program. This corresponds directly to the extra work done in the parallel program.

Data-local: the time the processor is stalled waiting for a data reference it issued to be satisfied by the memory system on its own processing node; that is, waiting for a reference that does not require communication with other nodes.

Data-remote: the time it is stalled waiting for data to be communicated to or from another (remote) processing node, whether due to inherent or artifactual communication. This represents the cost of communication as seen by the processor.

Synchronization: the time it spends waiting for another process to signal the occurrence of an event that will allow it to proceed. This includes the load imbalance and serialization in the program, as well as the time spent actually executing synchronization operations and accessing synchronization variables. While it is waiting, the processor could be repeatedly polling a variable until that variable changes value—thus executing instructions—or it could be stalled, depending on how synchronization is implemented.[1]

The *synchronization*, *busy-overhead* and *data-remote* components are not found in a sequential program running on a uniprocessor, and are overheads introduced by parallelism. As we have seen, while inherent communication is mostly included in the *data-remote* component, some (usually very small) part of it might show up as *data-local* time as well. For example, data that is assigned to the local memory of a processor P might be updated by another processor Q, but asynchronously returned to P's memory (due to replacement from Q, say) before P references it. Finally, the *data-local* component is interesting, since it is a performance overhead in both the sequential and parallel cases. While the other overheads tend to increase with the number of processors for a fixed problem, this component may decrease. This is because the processor is responsible for only a portion of the overall calculation, so it may only access a fraction of the data that the sequential program does and thus obtain better local cache and memory behavior. If the *data-local* overhead reduces enough, it can give rise to *superlinear* speedups even for deter-

---

1. A parallel algorithm is deterministic if the result it yields for a given input data set are always the same independent of the number of processes used or the relative timings of events. More generally, we may consider whether all the intermediate calculations in the algorithm are deterministic. A non-deterministic algorithm is one in which the result and the work done by the algorithm to arrive at the result depend on the number of processes and relative event timing. An example is a parallel search through a graph, which stops as soon as any path taken through the graph finds a solution. Non-deterministic algorithms complicate our simple model of where time goes, since the parallel program may do less useful work than the sequential program to arrive at the answer. Such situations can lead to *superlinear* speedup, i.e., speedup greater than the factor by which the number of processors is increased. However, not all forms of non-determinism have such beneficial results. Deterministic programs can also lead to superlinear speedups due to greater memory system overheads in the sequential program than a in a parallel execution, as we shall see.

1. Synchronization introduces components of time that overlap with other categories. For example, the time to satisfy the processor's first access to the synchronization variable for the current synchronization event, or the time spent actually communicating the occurrence of the synchronization event, may be included either in synchronization time or in the relevant data access category. We include it in the latter. Also, if a processor executes instructions to poll a synchronization variable while waiting for an event to occur, that time may be defined as busy-overhead or as synchronization. We include it in synchronization time, since it is essentially load imbalance.

ministic parallel programs. Figure 3-13 summarizes the correspondences between parallelization

Parallelization Step(s)	_Performance Issue_	_Processor Time Component_
Decomp./Assign./Orch.	Load Imbalance and Synchronization	Synch-wait
Decomp./Assign.	Extra Work	Busy-overhead
Decomp./Assign.	Inherent Comm. Volume	Data-remote
Orch.	Artifactual Comm. and Locality	Data-local
Orch./Mapping	Communication Structure	
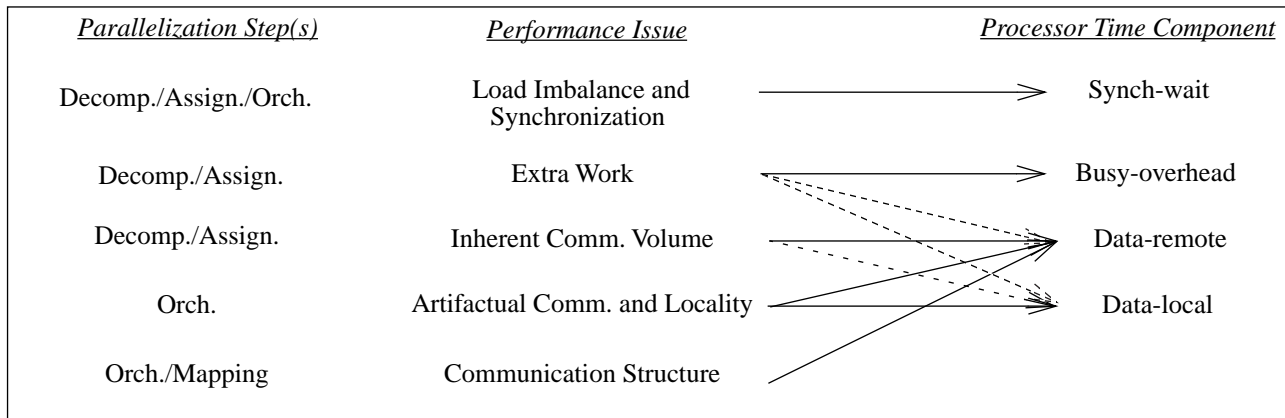
Figure 3-13 Mapping between parallelization issues and processor-centric components of execution time.

Bold lines depict direct relationships, while dotted lines depict side-effect contributions. On the left are shown the parallelization step in which the issues are mostly addressed.

issues, the steps in which they are mostly addressed, and processor-centric components of execution time.

Using these components, we may further refine our model of speedup for a fixed problem as follows, once again assuming a global synchronization at the end of the execution (otherwise we would take the maximum over processes in the denominator instead of taking the time profile of any single process):

$$\text{Speedup}_{prob}(p) \ = \ \frac{Busy(1) + Data_{\text{local}}(1)}{Busy_{useful}(p) + Data_{\text{local}}(p) + Synch(p) + Data_{\text{remote}}(p) + Busy_{overhead}(p)} \quad \textbf{(EQ 3.5)}$$

Our goal in addressing the performance issues has been to keep the terms in the denominator low and thus minimize the parallel execution time. As we have seen, both the programmer and the architecture have their roles to play. There is little the architecture can do to help if the program is poorly load balanced or if there is an inordinate amount of extra work. However, the architecture can reduce the incentive for creating such ill-behaved parallel programs by making communication and synchronization more efficient. The architecture can also reduce the artifactual communication incurred, provide convenient naming so that flexible assignment mechanisms can be easily employed, and make it possible to hide the cost of communication by overlapping it with useful computation.

3.6 The Parallel Application Case Studies: An In-Depth Look

Having discussed the major performance issues for parallel programs in a general context, and having applied them to the simple equation solver kernel, we are finally ready to examine what we really wanted to do all along in software: to achieve good parallel performance on more realistic applications on real multiprocessors. In particular, we are now ready to return to the four application case studies that motivated us to study parallel software in the previous chapter, apply

the four steps of the parallelization process to each case study, and at each step address the major performance issues that arise in it. In the process, we can understand and respond to the tradeoffs that arise among the different performance issues, as well as between performance and ease of programming. This will not only make our understanding of parallel software and tradeoffs more concrete, but will also help us see the types of workload characteristics that different applications present to a parallel architecture. Understanding the relationship between parallel applications, software techniques and workload characteristics will be very important as we go forward through the rest of the book.

Parallel applications come in various shapes and sizes, with very different characteristics and very different tradeoffs among the major performance issues. Our four case studies provide an interesting though necessarily very restricted cross-section through the application space. In examining how to parallelize, and particularly orchestrate, them for good performance, we shall focus for concreteness on a specific architectural style: a cache-coherent shared address space multiprocessor with main memory physically distributed among the processing nodes.

The discussion of each application is divided into four subsections. The first describes in more detail the sequential algorithms and the major data structures used. The second describes the partitioning of the application, i.e. the decomposition of the computation and its assignment to processes, addressing the algorithmic performance issues of load balance, communication volume and the overhead of computing the assignment. The third subsection is devoted to orchestration: it describes the spatial and temporal locality in the program, as well as the synchronization used and the amount of work done between synchronization points. The fourth discusses mapping to a network topology. Finally, for illustration we present the components of execution time as obtained for a real execution (using a particular problem size) on a particular machine of the chosen style: a 16-processor Silicon Graphics Origin2000. While the level of detail at which we treat the case studies may appear high in some places, these details will be important in explaining the experimental results we shall obtain in later chapters using these applications.

3.6.1 Ocean

Ocean, which simulates currents in an ocean basin, resembles many important applications in computational fluid dynamics. At each horizontal cross-section through the ocean basin, several different variables are modeled, including the current itself and the temperature, pressure, and friction. Each variable is discretized and represented by a regular, uniform two-dimensional grid of size $n+2$-by-$n+2$ points ($n+2$ is used instead of n so that the number of internal, non-border points that are actually computed in the equation solver is n-by-n). In all, about twenty-five different grid data structures are used by the application.

The Sequential Algorithm

After the currents at each cross-section are initialized, the outermost loop of the application proceeds over a large, user-defined number of time-steps. Every time-step first sets up and then solves partial differential equations on the grids. A time step consists of thirty three different grid computations, each involving one or a small number of grids (variables). Typical grid computations include adding together scalar multiples of a few grids and storing the result in another grid (e.g. $A = \alpha_1 B + \alpha_2 C - \alpha_3 D$), performing a single nearest-neighbor averaging sweep over a grid and storing the result in another grid, and solving a system of partial differential equations on a grid using an iterative method.

The iterative equation solver used is the multigrid method. This is a complex but efficient variant of the equation solver kernel we have discussed so far. In the simple solver, each iteration is a sweep over the entire n-by-n grid (ignoring the border columns and rows). A multigrid solver, on the other hand, performs sweeps over a hierarchy of grids. The original n-by-n grid is the finest-resolution grid in the hierarchy; the grid at each coarser level removes every alternate grid point in each dimension, resulting in grids of size $\frac{n}{2}-\text{by}-\frac{n}{2}$, $\frac{n}{4}-\text{by}-\frac{n}{4}$, and so on. The first sweep of the solver traverses the finest grid, and successive sweeps are performed on coarser or finer grids depending on the error computed in the previous sweep, terminating when the system converges within a user-defined tolerance. To keep the computation deterministic and make it more efficient, a red-black ordering is used (see Section 2.4.2 on page 105).

Decomposition and Assignment

Ocean affords concurrency at two levels within a time-step: across some grid computations (function parallelism), and within a grid computation (data parallelism). Little or no concurrency is available across successive time-steps. Concurrency across grid computations is discovered by writing down which grids each computation reads and writes, and analyzing the dependences among them at this level. The resulting dependence structure and concurrency are depicted in Figure 3-14. Clearly, there is not enough concurrency across grid computations—i.e. not enough vertical sections—to occupy more than a few processors. We must therefore exploit the data parallelism within a grid computation as well, and we need to decide what combination of function and data parallelism is best.

There are several reasons why we choose to have all processes collaborate on each grid computation, rather than divide the processes among the available concurrent grid computations and use both levels of parallelism; i.e. why we concentrate on data parallelism. Combined data and function parallelism would increase the size of each process's partition of a grid, and hence reduce communication-to-computation ratio. However, the work associated with different grid computations is very varied and depends on problem size in different ways, which complicates load balanced assignment. Second, since several different computations in a time-step access the same grid, for communication and data locality reasons we would not like the same grid to be partitioned in different ways among processes in different computations. Third, all the grid computations are fully data parallel and all grid points in a given computation do the same amount of work except at the borders, so we can statically assign grid points to processes using the data-parallel-only approach. Nonetheless, knowing which grid computations are independent is useful because it allows processes to avoid synchronizing between them.

The inherent communication issues are clearly very similar to those in the simple equation solver, so we use a block-structured domain decomposition of each grid. There is one complication—a tradeoff between locality and load balance related to the elements at the border of the entire grid. The internal n-by-n elements do similar work and are divided equally among all processes. Complete load balancing demands that border elements, which often do less work, also be divided equally among processors. However, communication and data locality suggest that border elements should be assigned to the processors that own the nearest internal elements. We follow the latter strategy, incurring a slight load imbalance.

Finally, let us examine the multigrid equation solver. The grids at all levels of the multigrid hierarchy are partitioned in the same block-structured domain decomposition. However, the number

[ARTIST: this figure is ugly in its current form. Anything you can do to beautify would be great. The idea is to have boxes that each represent a computation. Each row represents a computational phase. Boxes that are in the same horizontal row (phase) are independent of one another. The ones in the same vertical "column" are dependent, i.e. each depends on the one above it. I show these dependences by putting them in the same column, but the dependences could also be shown with arrows connecting the boxes (or ovals or whatever you choose to use). Showing the dependences among the computations is the main point of the figure.]

Put Laplacian of Ψ_1 in $W1_1$	Put Laplacian of Ψ_3 in $W1_3$	Copy Ψ_1, Ψ_3 into T_1, T_3	Put Ψ_1- Ψ_3 in W2	Put computed Ψ_2 values in W3	Initialize γ_a and γ_b
Add f values to columns of $W1_1$ and $W1_3$		Copy Ψ_{1M}, Ψ_{3M} into Ψ_1, Ψ_3			Put Laplacian of Ψ_{1M}, Ψ_{3M} in $W7_{1,3}$
Put Jacobians of ($W1_1$, T_1), ($W1_3$, T_3) in $W5_1$, $W5_3$		Copy T_1, T_3 into Ψ_{1M}, Ψ_{3M}			Put Laplacian of $W7_{1,3}$ in $W4_{1,3}$
			Put Jacobian of (W2, W3) in W6		Put Laplacian of $W4_{1,3}$ in $W7_{1,3}$
U P D A T E T H E γ E X P R E S S I O N S					
S O L V E T H E E Q U A T I O N F O R Ψ_a A N D P U T T H E R E S U L T I N γ_a					
C O M P U T E T H E I N T E G R A L O F Ψ_a					
Compute $\Psi = \Psi_a + C(t)\Psi_b$ (note: Ψ_a are maintained in γ_a matrix)			Solve the equation for Φ and put result in γ_b		
Use Ψ and Φ to update Ψ_1 and Ψ_3					
Update streamfunction running sums and determine whether to end program					

Note: Horizontal lines represent synchronization points among all processes, and vertical lines spanning phases demarcate threads of dependence.

Figure 3-14 Ocean: The phases in a time-step and the dependences among grid computations.

Each box is a grid computation. Computations in the same row are independent, while those in the same column are dependent.

of grid points per processor decreases as we go to coarser levels of the hierarchy. At the highest *log p* of the *log n* possible levels, there will be less grid points than the *p* processors, so some processors will remain idle. Fortunately, relatively little time is spent at these load-imbalanced levels. The ratio of communication to computation also increases at higher levels, since there are fewer points per processor. This illustrates the importance of measuring speedups relative to the best sequential algorithm (here multigrid, for example): A classical, non-hierarchical iterative solver on the original grid would likely yield better *self-relative* speedups (relative to a single processor performing the same computation) than the multigrid solver, but the multigrid solver is far more efficient sequentially and overall. In general, less efficient sequential algorithms often yield better self-relative "speedups"

Orchestration

Here we are mostly concerned with artifactual communication and data locality, and with the orchestration of synchronization. Let us consider issues related to spatial locality first, then temporal locality, and finally synchronization.

Spatial Locality: Within a grid computation, the issues related to spatial locality are very similar to those discussed for the simple equation solver kernel in Section 3.4.1, and we use four-dimensional array data structures to represent the grids. This results in very good spatial locality, partic-

ularly on local data. Accesses to nonlocal data (the elements at the boundaries of neighboring partitions) yield good spatial locality along row-oriented partition boundaries, and poor locality (hence fragmentation) along column-oriented boundaries. There are two major differences between the simple solver and the complete Ocean application in issues related to spatial locality. The first is that because Ocean involves thirty-three different grid computations in every time-step, each involving one or more out of twenty-five different grids, we experience many conflict misses *across* grids. While we alleviate this by ensuring that the dimensions of the arrays that we allocate are not powers of two (even if the program uses power-of-two grids), it is difficult to lay different grids out relative to one another to minimizes conflict misses. The second difference has to do with the multigrid solver. The fact that a process's partition has fewer grid points at higher levels of the grid hierarchy makes it more difficult to allocate data appropriately at page granularity and reduces spatial locality, despite the use of four-dimensional arrays.

Working Sets and Temporal Locality: Ocean has a complicated working set hierarchy, with six working sets. These first three are due to the use of near-neighbor computations, including the multigrid solver, and are similar to those for the simple equation solver kernel. The first is captured when the cache is large enough to hold a few grid elements, so that an element that is accessed as the right neighbor for another element is reused to compute itself and also as the left neighbor for the next element. The second working set comprises a couple of subrows of a process's partition. When the process returns from one subrow to the beginning of the next in a near-neighbor computation, it can reuse the elements of the previous subrow. The rest of the working sets are not well defined as single working sets, and lead to a curve without sharp knees. The third constitutes a process's entire partition of a grid used in the multigrid solver. This could be the partition at any level of the multigrid hierarchy at which the process tends to iterate, so it is not really a single working set. The fourth consists of the sum of a process's subgrids at several successive levels of the grid hierarchy within which it tends to iterate (in the extreme, this becomes all levels of the grid hierarchy). The fifth working set allows one to exploit reuse on a grid across grid computations or even phases; thus, it is large enough to hold a process's partition of several grids. The last holds all the data that a process is assigned in every grid, so that all these data can be reused across times-steps.

The working sets that are most important to performance are the first three or four, depending on how the multigrid solver behaves. The largest among these grow linearly with the size of the data set *per process*. This growth rate is common in scientific applications that repeatedly stream through their data sets, so with large problems some important working sets do not fit in the local caches. Note that with proper data placement the working sets for a process consist mostly of local rather than communicated data. The little reuse that nonlocal data afford is captured by the first two working sets.

Synchronization: Ocean uses two types of synchronization. First, global barriers are used to synchronize all processes between computational phases (the horizontal lines in Figure 3-14), as well between iterations of the multigrid equation solver. Between several of the phases we could replace the barriers with finer-grained point-to-point synchronization at element level to obtain some overlap across phases; however, the overlap is likely to be too small to justify the overhead of many more synchronization operations and the programming complexity. Second, locks are used to provide mutual exclusion for global reductions. The work between synchronization points is very large, typically proportional to the size of a processor's partition of a grid.

Mapping

Given the near-neighbor communication pattern, we would like to map processes to processors such that processes whose partitions are adjacent to each other in the grid run on processors that are adjacent to each other in the network topology. Our subgrid partitioning of two-dimensional grids clearly maps very well to a two-dimensional mesh network.

(a) Four-dimensional arrays

(a) Two-dimensional arrays
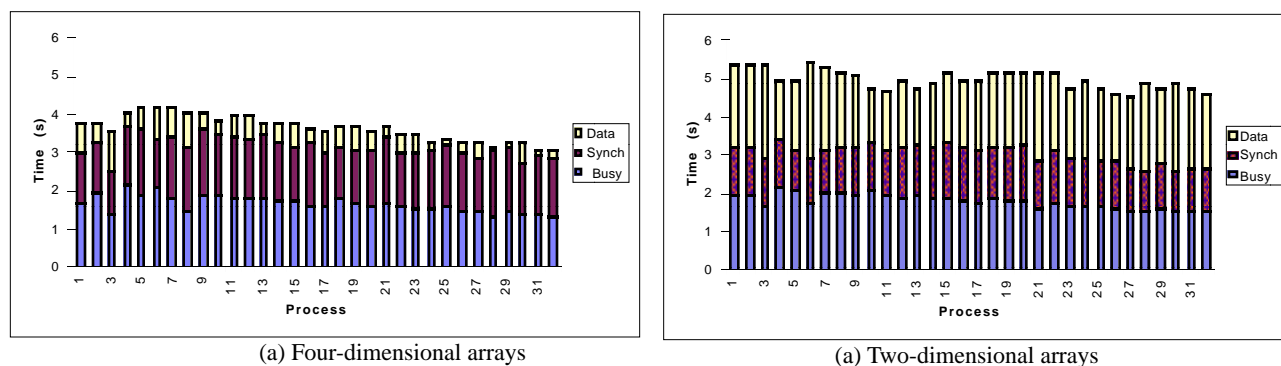
Figure 3-15 Execution time breakdowns for Ocean on a 32-processor Origin2000.

The size of each grid is 1030-by-1030, and the convergence tolerance is 10^{-3}. The use of four-dimensional arrays to represent the two-dimensional arrays to represent the two-dimensional grids clearly reduces the time spent stalled on the memory system (including communication). This data wait time is very small because a processor's partition of the grids it uses at a time fit very comfortably in the large 4MB second-level caches in this machine. With smaller caches, or much bigger grids, the time spent stalled waiting for (local) data would have been much larger.

In summary, Ocean is a good representative of many computational fluid dynamics computations that use regular grids. The computation to communication ratio is proportional to $\frac{n}{\sqrt{p}}$ for a problem with *n*-by-*n* grids and *p* processors, load balance is good except when *n* is not large relative to *p*, and the parallel efficiency for a given number of processors increases with the grid size. Since a processor streams through its portion of the grid in each grid computation, since only a few instructions are executed per access to grid data during each sweep, and since there is significant potential for conflict misses across grids, data distribution in main memory can be very important on machines with physically distributed memory.

Figure 3-15 shows the breakdown of execution time into busy, waiting at synchronization points, and waiting for data accesses to complete for a particular execution of Ocean with 1030-by-1030 grids on a 32-processor SGI Origin2000 machine. As in all our programs, mapping of processes to processors is not enforced by the program but is left to the system. This machine has very large per-processor second-level caches (4MB), so with four-dimensional arrays each processor's partition tends to fit comfortably in its cache. The problem size is large enough relative to the number of processors that the inherent communication to computation ratio is quite low. The major bottleneck is the time spent waiting at barriers. Smaller problems would stress communication more, while larger problems and proper data distribution would put more stress on the local memory system. With two-dimensional arrays, the story is clearly different. Conflict misses are frequent, and with data being difficult to place appropriately in main memory, many of these misses are not satisfied locally, leading to long latencies as well as contention.

3.6.2 Barnes-Hut

The galaxy simulation has far more irregular and dynamically changing behavior than Ocean. The algorithm it uses for computing forces on the stars, the Barnes-Hut method, is an efficient hierarchical method for solving the n-body problem in *O(n log n)* time. Recall that the n-body problem is the problem of computing the influences that *n* bodies in a system exert on one another.

The Sequential Algorithm

The galaxy simulation proceeds over hundreds of time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. Recall the insight
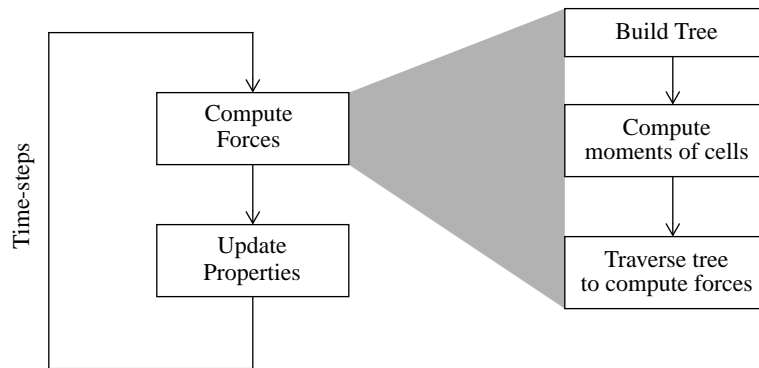
Figure 3-16 Flow of computation in the Barnes-Hut application.

that force calculation in the Barnes-Hut method is based on: If the magnitude of interaction between bodies falls off rapidly with distance (as it does in gravitation), then the effect of a large group of bodies may be approximated by a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated. The hierarchical application of this insight implies that the farther away the bodies, the larger the group that can be approximated by a single body.

To facilitate a hierarchical approach, the Barnes-Hut algorithm represents the three-dimensional space containing the galaxies as a tree, as follows. The root of the tree represents a space cell containing all bodies in the system. The tree is built by adding bodies into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a fixed number of bodies (here ten). The result is an "oct-tree" whose internal nodes are cells and whose leaves are individual bodies.[1] Empty cells resulting from a cell subdivision are ignored. The tree, and the Barnes-Hut algorithm, is therefore adaptive in that it extends to more levels in regions that have high body densities. While we use a three-dimensional problem, Figure 3-17 shows a small

---

1. An oct-tree is a tree in which every node has a maximum of eight children. In two dimensions, a quadtree would be used, in which the maximum number of children is four.

two-dimensional example domain and the corresponding "quadtree" for simplicity. The positions of the bodies change across time-steps, so the tree has to be rebuilt every time-step. This results in the overall computational structure shown in the right hand side of Figure 3-16, with most of the time being spent in the force calculation phase.

(a)   The Spatial Domain   (b)   Quadtree Representation
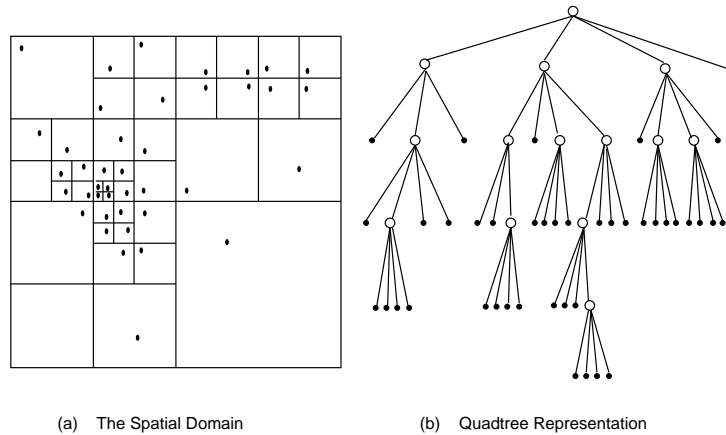
Figure 3-17 Barnes-Hut: A two-dimensional particle distribution and the corresponding quadtree.

The tree is traversed once per body to compute the net force acting on that body. The force-calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single body at the center of mass of the cell, and the force this center of mass exerts on the body computed. If, however, the center of mass is not far enough away, the cell must be "opened" and each of its subcells visited. A cell is determined to be far enough away if the following condition is satisfied:

$$\frac{l}{d} < \theta,$$

(EQ 3.6)

where l is the length of a side of the cell, d is the distance of the body from the center of mass of the cell, and θ is a user-defined accuracy parameter (θ is usually between 0.5 and 1.2). In this way, a body traverses deeper down those parts of the tree which represent space that is physically close to it, and groups distant bodies at a hierarchy of length scales. Since the expected depth of the tree is $O(\log n)$, and the number of bodies for which the tree is traversed is n, the expected complexity of the algorithm is $O(n \log n)$. Actually it is $O(\frac{1}{\theta^2} n \log n)$, since θ determines the number of tree cells touched at each level in a traversal.

Principal Data Structures

Conceptually, the main data structure in the application is the Barnes-Hut tree. The tree is implemented in both the sequential and parallel programs with two arrays: an array of bodies and an array of tree cells. Each body and cell is represented as a structure or record. The fields for a body include its three-dimensional position, velocity and acceleration, as well as its mass. A cell structure also has pointers to its children in the tree, and a three-dimensional center-of-mass. There is also a separate array of pointers to bodies and one of pointers to cells. Every process owns an equal contiguous chunk of pointers in these arrays, which in every time-step are set to point to the

bodies and cells that are assigned to it in that time-step. The structure and partitioning of the tree changes across time-steps as the galaxy evolves, the actual bodies and cells assigned to a process are not contiguous in the body and cell arrays.

Decomposition and Assignment

Each of the phases within a time-step is executed in parallel, with global barrier synchronization between phases. The natural unit of decomposition (task) in all phases is a body, except in computing the cell centers of mass, where it is a cell.

Unlike Ocean, which has a regular and predictable structure of both computation and communication, the Barnes-Hut application presents many challenges for effective assignment. First, the non-uniformity of the galaxy implies that the amount of work per body and the communication patterns are nonuniform, so a good assignment cannot be discovered by inspection. Second, the distribution of bodies changes across time-steps, which means that no static assignment is likely to work well. Third, since the information needs in force calculation fall off with distance equally in all directions, reducing interprocess communication demands that partitions be spatially contiguous and not biased in size toward any one direction. And fourth, the different phases in a time-step have different distributions of work among the bodies/cells, and hence different preferred partitions. For example, the work in the update phase is uniform across all bodies, while that in the force calculation phase clearly is not. Another challenge for good performance is that the communication needed among processes is naturally fine-grained and irregular.

We focus our partitioning efforts on the force-calculation phase, since it is by far the most time-consuming. The partitioning is not modified for other phases since (a) the overhead of doing so (both in partitioning and in the loss of locality) outweighs the potential benefits, and (b) similar partitions are likely to work well for tree building and moment calculation (though not for the update phase).

As we have mentioned earlier in the chapter, we can use profiling-based semi-static partitioning in this application, taking advantage of the fact that although the particle distribution at the end of the simulation may be radically different from that at the beginning, it evolves slowly with time does not change very much between two successive time-steps. As we perform the force calculation phase in a time-step, we record the work done by every particle in that time-step (i.e. count the number of interactions it computes with other bodies or cells). We then use this work count as a measure of the work associated with that particle in the next time-step. Work counting is very cheap, since it only involves incrementing a local counter when an (expensive) interaction is performed. Now we need to combine this load balancing method with assignment techniques that also achieve the communication goal: keeping partitions contiguous in space and not biased in size toward any one direction. We briefly discuss two techniques: the first because it is applicable to many irregular problems and we shall refer to it again in Section 3.7; the second because it is what our program uses.

The first technique, called orthogonal recursive bisection (ORB), preserves physical locality by partitioning the domain space directly. The space is recursively subdivided into two subspaces with equal cost, using the above load balancing measure, until there is one subspace per process (see Figure 3-18(a)). Initially, all processes are associated with the entire domain space. Every time a space is divided, half the processes associated with it are assigned to each of the subspaces that result. The Cartesian direction in which division takes place is usually alternated with suc-

cessive divisions, and a parallel median finder is used to determine where to split the current sub-space. A separate binary tree of depth *log p* is used to implement ORB. Details of using ORB for this application can be found in [Sal90].

| (a) ORB | (b) Costzones |
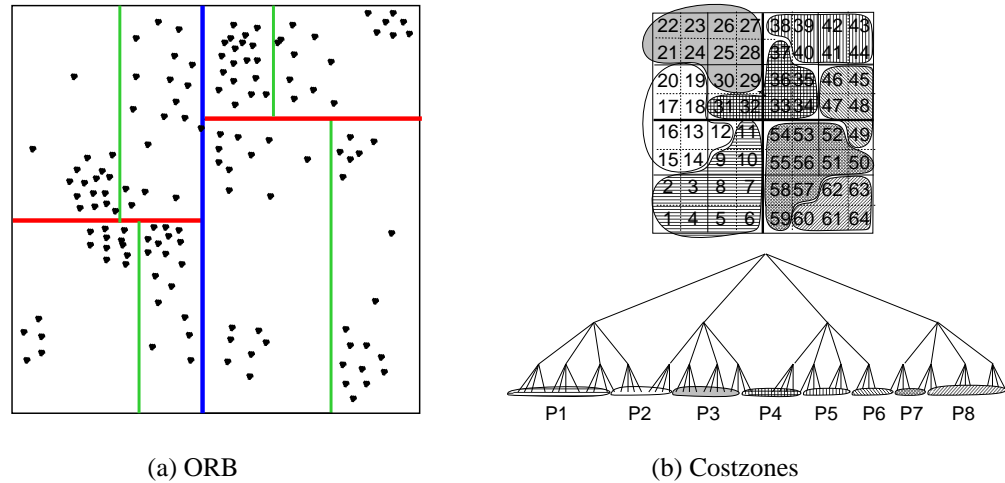
Figure 3-18 Partitioning schemes for Barnes-Hut: ORB and Costzones.

ORB partitions space directly by recursive bisection, while costzones partitions the tree. (b) shows both the partitioning of the tree as well as how the resulting space is partitioned by costzones. Note that ORB leads to more regular (rectangular) partitions than cost-zones.

The second technique, called costzones, takes advantage of the fact that the Barnes-Hut algorithm already has a representation of the spatial distribution of bodies encoded in its tree data structure. Thus, we can partition this existing data structure itself and obtain the goal of partitioning space (see Figure 3-18(b)). Here is a high-level description. Every internal cell stores the total cost associated with all the bodies it contains. The total work or cost in the system is divided among processes so that every process has a contiguous, equal range or zone of work (for example, a total work of 1000 units would be split among 10 processes so that zone 1-100 units is assigned to the first process, zone 101-200 to the second, and so on). Which cost zone a body in the tree belongs to can be determined by the total cost of an inorder traversal of the tree up to that body. Processes traverse the tree in parallel, picking up the bodies that belong in their cost zone. Details can be found in [SH+95]. Costzones is much easier to implement than ORB. It also and yields better overall performance in a shared address space, mostly because the time spent in the partitioning phase itself is much smaller, illustrating the impact of extra work.

Orchestration

Orchestration issues in Barnes-Hut reveal many differences from Ocean, illustrating that even applications in scientific computing can have widely different behavioral characteristics of architectural interest.

Spatial Locality: While the shared address space makes it easy for a process to access the parts of the shared tree that it needs in all the computational phases (see Section 3.7), data distribution to keep a process's assigned bodies and cells in its local main memory is not so easy as in Ocean.

First, data have to be redistributed dynamically as assignments change across time-steps, which is expensive. Second, the logical granularity of data (a particle/cell) is much smaller than the physical granularity of allocation (a page), and the fact that bodies/cells are spatially contiguous in the arrays does not mean they are contiguous in physical space or assigned to the same process. Fixing these problems requires overhauling the data structures that store bodies and cells: using separate arrays or lists per process, that are modified across time-steps. Fortunately, there is enough temporal locality in the application that data distribution is not so important in a shared address space (again unlike Ocean). Also, the vast majority of the cache misses are to data in other processors' assigned partitions anyway, so data distribution itself wouldn't help make them local. We therefore simply distribute pages of shared data in a round-robin interleaved manner among nodes, without attention to which node gets which pages.

While in Ocean long cache blocks improve local access performance limited only by partition size, here multi-word cache blocks help exploit spatial locality only to the extent that reading a particle's displacement or moment data involves reading several double-precision words of data. Very long transfer granularities might cause more fragmentation than useful prefetch, for the same reason that data distribution at page granularity is difficult: Unlike Ocean, locality of bodies/cells in the data structures does not match that in physical space on which assignment is based, so fetching data from more than one particle/cell upon a miss may be harmful rather than beneficial.

Working Sets and Temporal Locality: The first working set in this program contains the data used to compute forces between a single particle-particle or particle-cell pair. The interaction with the next particle or cell in the traversal will reuse these data. The second working set is the most important to performance. It consists of the data encountered in the entire tree traversal to compute the force on a single particle. Because of the way partitioning is done, the traversal to compute the forces on the next particle will reuse most of these data. As we go from particle to particle, the composition of this working set changes slowly. However, the amount of reuse is tremendous, and the resulting working set is small even though overall a process accesses a very large amount of data in irregular ways. Much of the data in this working set is from other processes' partitions, and most of these data are allocated nonlocally. Thus, it is the temporal locality exploited on shared (both local and nonlocal) data that is critical to the performance of the application, unlike Ocean where it is data distribution.

By the same reasoning that the complexity of the algorithm is $O(\frac{1}{\theta^2} n \log n)$, the expected size of this working set is proportional to $O(\frac{1}{\theta^2} \log n)$, even though the overall memory requirement of the application is close to linear in n: Each particle accesses about this much data from the tree to compute the force on it. The constant of proportionality is small, being the amount of data accessed from each body or cell visited during force computation. Since this working set fits comfortably in modern second-level caches, we do not replicate data in main memory. In Ocean there were important working sets that grew linearly with the data set size, and we did not always expect them to fit in the cache; however, even there we did not need replication in main memory since if data were distributed appropriately then the data in these working sets were local.

Synchronization: Barriers are used to maintain dependences among bodies and cells across some of the computational phases, such as between building the tree and using it to compute forces. The unpredictable nature of the dependences makes it much more difficult to replace the barriers by point-to-point body or cell level synchronization. The small number of barriers used in a time-step is independent of problem size or number of processors.

There is no need for synchronization within the force computation phase itself. While communication and sharing patterns in the application are irregular, they are phase-structured. That is, while a process reads particle and cell data from many other processes in the force calculation phase, the fields of a particle structure that are written in this phase (the accelerations and velocities) are not the same as those that are read in it (the displacements and masses). The displacements are written only at the end of the update phase, and masses are not modified. However, in other phases, the program uses both mutual exclusion with locks and point-to-point event synchronization with flags in more interesting ways than Ocean. In the tree building phase, a process that is ready to add a particle to a cell must first obtain mutually exclusive access to the cell, since other processes may want to read or modify the cell at the same time. This is implemented with a lock per cell. The moment calculation phase is essentially an upward pass through the tree from the leaves to the root, computing the moments of cells from those of their children. Point-to-point event synchronization is implemented using flags to ensure that a parent does not read the moment of its child until that child has been updated by all its children. This is an example of multiple-producer, single-consumer group synchronization. There is no synchronization within the update phase.

The work between synchronization points is large, particularly in the force computation and update phases, where it is $O\left(\frac{n\log n}{p}\right)$ and $O(\frac{n}{p})$, respectively. The need for locking cells in the tree-building and center-of-mass phases causes the work between synchronization points in those phases to be substantially smaller.

Mapping

The irregular nature makes this application more difficult to map perfectly for network locality in common networks such as meshes. The ORB partitioning scheme maps very naturally to a hypercube topology (discussed in Chapter 10), but not so well to a mesh or other less richly interconnected network. This property does not hold for costzones partitioning, which naturally maps to a one-dimensional array of processors but does not easily guarantee to keep communication local even in such a network.

In summary, the Barnes-Hut application has irregular, fine-grained, time-varying communication and data access patterns that are becoming increasingly prevalent even in scientific computing as we try to model more complex natural phenomena. Successful partitioning techniques for it are not obvious by inspection of the code, and require the use of insights from the application domain. These insights allow us to avoid using fully dynamic assignment methods such as task queues and stealing.

Figure 3-19 shows the breakdowns of execution time for this application. Load balance is quite good with a static partitioning of the array of bodies to processors, precisely because there is little relationship between their location in the array and in physical space. However, the data access cost is high, since there is a lot of inherent and artifactual communication. Semi-static, costzones partitioning reduces this data access overhead substantially without compromising load balance.

3.6.3 Raytrace

Recall that in ray tracing rays are shot through the pixels in an image plane into a three-dimensional scene, and the paths of the rays traced as they bounce around to compute a color and opac-

(a) Static assignment of bodies

(a) Semi-static, costzones assignment
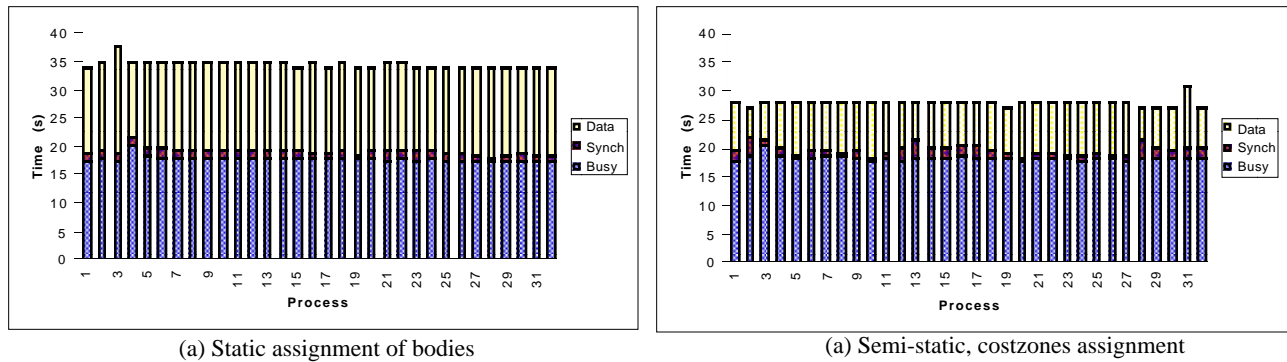
Figure 3-19 Execution time breakdown for Barnes-Hut with 512K bodies on the Origin2000.

The particular static assignment of bodies used is quite randomized, so given the large number of bodies relative to processors the workload evens out due to the law of large numbers. The bigger problem with the static assignment is that because it is effectively randomized the particles assigned to a processor are not close together in space so the communication to computation ratio is much larger. This is why data wait time is much smaller in the semi-static scheme. If we had assigned contiguous areas of space to processes statically, data wait time would be small but load imbalance and hence synchronization wait time would be large. Even with the current static assignment, there is no guarantee that the assignment will remain load balanced as the galaxy evolves over time.

ity for the corresponding pixels. The algorithm uses a hierarchical representation of space called a Hierarchical Uniform Grid (HUG), which is similar in structure to the octree used by the Barnes-Hut application. The root of the tree represents the entire space enclosing the scene, and the leaves hold a linked list of the object primitives that fall in them (the maximum number of primitives per leaf is also defined by the user). The hierarchical grid or tree makes it efficient to skip empty regions of space when tracing a ray, and quickly find the next interesting cell.

Sequential Algorithm

For a given viewpoint, the sequential algorithm fires one ray into the scene through every pixel in the image plane. These initial rays are called primary rays. At the first object that a ray encounters (found by traversing the hierarchical uniform grid), it is first reflected toward every light source to determine whether it is in shadow from that light source. If it isn't, the contribution of the light source to its color and brightness is computed. The ray is also reflected from and refracted through the object as appropriate. Each reflection and refraction spawns a new ray, which undergoes the same procedure recursively for every object that it encounters. Thus, each primary ray generates a tree of rays. Rays are terminated either when they leave the volume enclosing the scene or according to some user-defined criterion (such as the maximum number of levels allowed in a ray tree). Ray tracing, and computer graphics in general, affords several tradeoffs between execution time and image quality, and many algorithmic optimizations have been developed to improve performance without compromising image quality much.

Decomposition and Assignment

There are two natural approaches to exploiting parallelism in ray tracing. One is to divide the space and hence the objects in the scene among processes, and have a process compute the interactions for rays that occur within its space. The unit of decomposition here is a subspace. When a ray leaves a process's subspace, it will be handled by the next process whose subspace it enters. This is called a *scene-oriented* approach. The alternate, *ray-oriented* approach is to divide pixels

in the image plane among processes. A process is responsible for the rays that are fired through its assigned pixels, and follows a ray in its path through the entire scene, computing the interactions of the entire ray tree which that ray generates. The unit of decomposition here is a primary ray. It can be made finer by allowing different processes to process rays generated by the same primary ray (i.e. from the same ray tree) if necessary. The scene-oriented approach preserves more locality in the scene data, since a process only touches the scene data that are in its subspace and the rays that enter that subspace. However, the ray-oriented approach is much easier to implement with low overhead, particularly starting from a sequential program, since rays can be processed independently without synchronization and the scene data are read-only. This program therefore uses a ray-oriented approach. The degree of concurrency for a n-by-n plane of pixels is $O(n^2)$, and is usually ample.

Unfortunately, a static subblock partitioning of the image plane would not be load balanced. Rays from different parts of the image plane might encounter very different numbers of reflections and hence very different amounts of work. The distribution of work is highly unpredictable, so we use a distributed task queueing system (one queue per processor) with task stealing for load balancing.

Consider communication. Since the scene data are read-only, there is no inherent communication on these data. If we replicated the entire scene on every node, there would be no communication except due to task stealing. However this approach does not allow us to render a scene larger than what fits in a single processor's memory. Other than task stealing, communication is generated because only $1/p$ of the scene is allocated locally and a process accesses the scene widely and unpredictably. To reduce this artifactual communication, we would like processes to reuse scene data as much as possible, rather than access the entire scene randomly. For this, we can exploit spatial coherence in ray tracing: Because of the way light is reflected and refracted, rays that pass through adjacent pixels from the same viewpoint are likely to traverse similar parts of the scene and be reflected in similar ways. This suggests that we should use domain decomposition on the image plane to assign pixels to task queues initially. Since the adjacency or spatial coherence of rays works in all directions in the image plane, block-oriented decomposition works well. This also reduces the communication of image pixels themselves.

Given p processors, the image plane is partitioned into p rectangular blocks of size as close to equal as possible. Every image block or partition is further subdivided into fixed sized square image *tiles*, which are the units of task granularity and stealing (see Figure 3-20 for a four-process example). These tile tasks are initially inserted into the task queue of the processor that is assigned that block. A processor ray traces the tiles in its block in scan-line order. When it is done with its block, it steals tile tasks from other processors that are still busy. The choice of tile size is a compromise between preserving locality and reducing the number of accesses to other processors' queues, both of which reduce communication, and keeping the task size small enough to ensure good load balance. We could also initially assign tiles to processes in an interleaved manner in both dimensions (called a *scatter decomposition*) to improve load balance in the initial assignment.

Orchestration

Given the above decomposition and assignment, let us examine spatial locality, temporal locality, and synchronization.
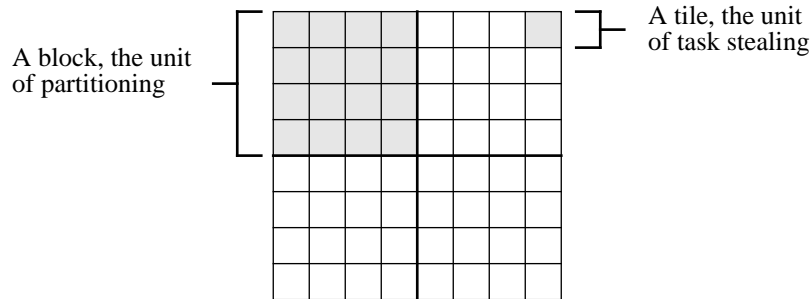
Figure 3-20 Image plane partitioning in Raytrace for four processors.

Spatial Locality: Most of the shared data accesses are to the scene data. However, because of changing viewpoints and the fact that rays bounce about unpredictably, it is impossible to divide the scene into parts that are each accessed only (or even dominantly) by a single process. Also, the scene data structures are naturally small and linked together with pointers, so it is very difficult to distribute them among memories at the granularity of pages. We therefore resort to using a round-robin layout of the pages that hold scene data, to reduce contention. Image data are small, and we try to allocate the few pages they fall on in different memories as well. The sub-block partitioning described above preserves spatial locality at cache block granularity in the image plane quite well, though it can lead to some false sharing at tile boundaries, particularly with task stealing. A strip decomposition in rows of the image plane would be better from the viewpoint of spatial locality, but would not exploit spatial coherence in the scene as well. Spatial locality on scene data is not very high, and does not improve with larger scenes.

Temporal Locality: Because of the read-only nature of the scene data, if there were unlimited capacity for replication then only the first reference to a nonlocally allocated datum would cause communication. With finite replication capacity, on the other hand, data may be replaced and have to be recommunicated. The domain decomposition and spatial coherence methods described earlier enhance temporal locality on scene data and reduce the sizes of the working sets. However, since the reference patterns are so unpredictable due to the bouncing of rays, working sets are relatively large and ill-defined. Note that most of the scene data accessed and hence the working sets are likely to be nonlocal. Nonetheless, this shared address space program does not replicate data in main memory: The working sets are not sharp and replication in main memory has a cost, so it is unclear that the benefits outweigh the overheads.

Synchronization and Granularity: There is only a single barrier after an entire scene is rendered and before it is displayed. Locks are used to protect task queues for task stealing, and also for some global variables that track statistics for the program. The work between synchronization points is the work associated with tiles of rays, which is usually quite large.

Mapping

Since Raytrace has very unpredictable access and communication patterns to scene data, the communication is all but impossible to map effectively in this shared address space version. The fact that the initial assignment of rays partitions the image into a two-dimensional grid of blocks, it would be natural to map to a two-dimensional mesh network but the effect is not likely to be large.

| (a) With task stealing | (a) Without task stealing |
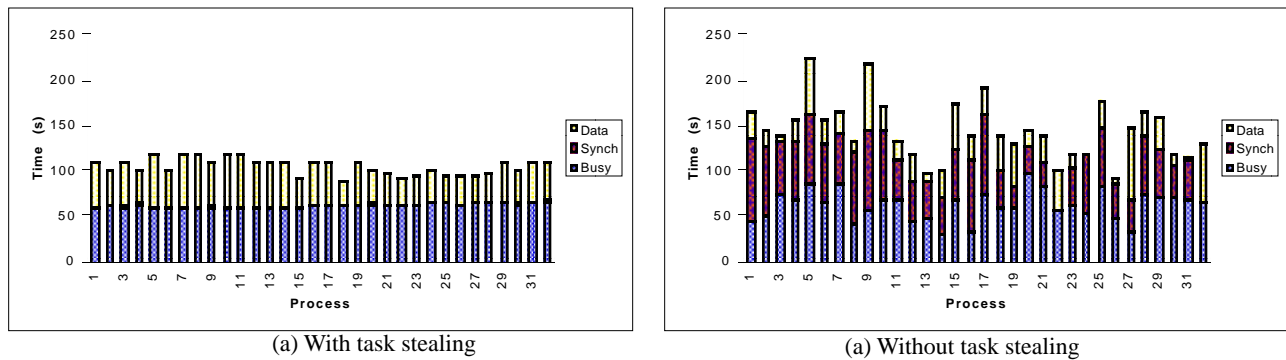
Figure 3-21 Execution time breakdowns for Raytrace with the balls data set on the Origin2000.

Task stealing is clearly very important for balancing the workload (and hence reducing synchronization wait time) in this highly unpredictable application.

In summary, this application tends to have large working sets and relatively poor spatial locality, but a low inherent communication to computation ratio. Figure 3-21 shows the breakdown of execution time for the balls data set, illustrating the importance of task stealing in reducing load imbalance. The extra communication and synchronization incurred as a result is well worthwhile.

3.6.4 Data Mining

A key difference in the data mining application from the previous ones is that the data being accessed and manipulated typically reside on disk rather than in memory. It is very important to reduce the number of disk accesses, since their cost is very high, and also to reduce the contention for a disk controller by different processors.

Recall the basic insight used in association mining from Section 2.2.4: If an itemset of size k is large, then all subsets of that itemset must also be large. For illustration, consider a database in which there are five items—A, B, C, D, and E—of which one or more may be present in a particular transaction. The items within a transaction are lexicographically sorted. Consider L_2, the list of large itemsets of size 2. This list might be {AB, AC, AD, BC, BD, CD, DE}. The itemsets within L_2 are also lexicographically sorted. Given this L_2, the list of itemsets that are candidates for membership in L_3 are obtained by performing a "join" operation on the itemsets in L_2; i.e. taking pairs of itemsets in L_2 that share a common first item (say AB and AC), and combining them into a lexicographically sorted 3-itemset (here ABC). The resulting candidate list C_3 in this case is {ABC, ABD, ACD, BCD}. Of these itemsets in C_3, some may actually occur with enough frequency to be placed in L_3, and so on. In general, the join to obtain C_k from L_{k-1} finds pairs of itemsets in L_{k-1} whose first $k-2$ items are the same, and combines them to create a new item for C_k. Itemsets of size $k-1$ that have common $k-2$ sized prefixes are said to form an equivalence class (e.g. {AB, AC, AD}, {BC, BD}, {CD} and {DE} in the example above). Only itemsets in the same $k-2$ equivalence class need to be considered together to form C_k from L_{k-1}, which greatly reduces the number of pairwise itemset comparisons we need to do to determine C_k.

Sequential Algorithm

A simple sequential method for association mining is to first traverse the dataset and record the frequencies of all itemsets of size one, thus determining L_1. From L_1, we can construct the candidate list C_2, and then traverse the dataset again to find which entries of C_2 are large and should be in L_2. From L_2, we can construct C_3, and traverse the dataset to determine L_3, and so on until we have found L_k. While this method is simple, it requires reading all transactions in the database from disk k times, which is expensive.

The key goals in a sequential algorithm are to reduce the amount of work done to compute candidate lists C_k from lists of large itemsets L_{k-1}, and especially to reduce the number of times data must be read from disk in determining the counts of itemsets in a candidate list C_k (to determine which itemsets should be in L_k). We have seen that equivalence classes can be used to achieve the first goal. In fact, they can be used to construct a method that achieves both goals together. The idea is to transform the way in which the data are stored in the database. Instead of storing transactions in the form $\{T_x, A, B, D, ...\}$—where T_x is the transaction identifier and A, B, D are items in the transaction—we can keep in the database records of the form $\{IS_x, T1, T2, T3, ...\}$, where IS_x is an itemset and T1, T2 etc. are transactions that contain that itemset. That is, there is a database record per itemset rather than per transaction. If the large itemsets of size k-1 (L_{k-1}) that are in the same *k-2* equivalence class are identified, then computing the candidate list C_k requires only examining all pairs of these itemsets. If each itemset has its list of transactions attached to it, as in the above representation, then the size of each resulting itemset in C_k can be computed at the same time as identifying the C_k itemset itself from a pair of L_{k-1} itemsets, by computing the intersection of the transactions in their lists.

For example, suppose {AB, 1, 3, 5, 8, 9} and {AC, 2, 3, 4, 8, 10} are large 2-itemsets in the same 1-equivalence class (they each start with A), then the list of transactions that contain itemset ABC is {3, 8}, so the occurrence count of itemset ABC is two. What this means is that once the database is transposed and the 1-equivalence classes identified, the rest of the computation for a single 1-equivalence class can be done to completion (i.e. all large k-itemsets found) before considering any data from other 1-equivalence classes. If a 1-equivalence class fits in main memory, then after the transposition of the database a given data item needs to be read from disk only once, greatly reducing the number of expensive I/O accesses.

Decomposition and Assignment

The two sequential methods also differ in their parallelization, with the latter method having advantages in this respect as well. To parallelize the first method, we could first divide the database among processors. At each step, a processor traverses only its local portion of the database to determine partial occurrence counts for the candidate itemsets, incurring no communication or nonlocal disk accesses in this phase. The partial counts are then merged into global counts to determine which of the candidates are large. Thus, in parallel this method requires not only multiple passes over the database but also requires interprocessor communication and synchronization at the end of every pass.

In the second method, the equivalence classes that helped the sequential method reduce disk accesses are very useful for parallelization as well. Since the computation on each 1-equivalence class is independent of the computation on any other, we can simply divide up the 1-equivalence classes among processes which can thereafter proceed independently, without communication or

synchronization. The itemset lists (in the transformed format) corresponding to an equivalence class can be stored on the local disk of the process to which the equivalence class is assigned, so there is no need for remote disk access after this point either. As in the sequential algorithm, since each process can complete the work on one of its assigned equivalence classes before proceeding to the next one, hopefully each item from the local database should be read only once. The issue is ensuring a load balanced assignment of equivalence classes to processes. A simple metric for load balance is to assign equivalence classes based on the number of initial entries in them. However, as the computation unfolds to compute k-itemsets, the amount of work is determined more closely by the number of large itemsets that are generated at each step. Heuristic measures that estimate this or some other more appropriate work metric can be used as well. Otherwise, one may have to resort to dynamic tasking and task stealing, which can compromise much of the simplicity of this method (e.g. that once processes are assigned their initial equivalence classes they do not have to communicate, synchronize, or perform remote disk access).

The first step in this approach, of course, is to compute the 1-equivalence classes and the large 2-itemsets in them, as a starting point for the parallel assignment. To compute the large 2-itemsets, we are better off using the original form of the database rather than the transformed form, so we do not transform the database yet (see Exercise 3.12). Every process sweeps over the transactions in its local portion of the database, and for each pair of items in a transaction increments a local counter for that item pair (the local counts can be maintained as a two-dimensional upper-triangular array, with the indices being items). It is easy to compute 2-itemset counts directly, rather than first make a pass to compute 1-itemset counts, construct the list of large 1-itemsets, and then make another pass to compute 2-itemset counts. The local counts are then merged, involving interprocess communication, and the large 2-itemsets determined from the resulting global counts. These 2-itemsets are then partitioned into 1-equivalence classes, and assigned to processes as described above.

The next step is to transform the database from the original $\{T_x, A, B, D, ...\}$ organization by transaction to the $\{IS_x, T1, T2, T3, ...\}$ organization by itemset, where the IS_x are initially the 2-itemsets. This can be done in two steps: a local step and a communication step. In the local step, a process constructs the partial transaction lists for large 2-itemsets from its local portion of the database. Then, in the communication step a process (at least conceptually) "sends" the lists for those 2-itemsets whose 1-equivalence classes are not assigned to it to the process to which they are assigned, and "receives" from other processes the lists for the equivalence classes that are assigned to it. The incoming partial lists are merged into the local lists, preserving a lexicographically sorted order, after which the process holds the transformed database for its assigned equivalence classes. It can now compute the k-itemsets step by step for each of its equivalence classes, without any communication, synchronization or remote disk access. The communication step of the transformation phase is usually the most expensive step in the algorithm. Finally, the results for the large k-itemsets are available from the different processes.

Orchestration

Given this decomposition and assignment, let us examine spatial locality, temporal locality, and synchronization.

**Spatial Locality**: Given the organization of the computation and the lexicographic sorting of the itemsets and transactions, most of the traversals through the data are simple front-to-back sweeps and hence exhibit very good predictability and spatial locality. This is particularly important in

reading from disk, since it is important to amortize the high startup costs of a disk read over a large amount of useful data read.

Temporal Locality: Proceeding over one equivalence class at a time is much like blocking, although how successful it is depends on whether the data for that equivalence class fit in main memory. As the computation for an equivalence class proceeds, the number of large itemsets becomes smaller, so reuse in main memory is more likely to be exploited. Note that here we are exploiting temporal locality in main memory rather than in the cache, although the techniques and goals are similar.

Synchronization: The major forms of synchronization are the reductions of partial occurrence counts into global counts in the first step of the algorithm (computing the large 2-itemsets), and a barrier after this to begin the transformation phase. The reduction is required only for 2-itemsets, since thereafter every process proceeds independently to compute the large k-itemsets in its assigned equivalence classes. Further synchronization may be needed if dynamic task management is used for load balancing.

This concludes our in-depth discussion of how the four case studies might actually be parallelized. At least in the first three, we assumed a coherent shared address space programming model in the discussion. Let us now see how the characteristics of these case studies and other applications influence the tradeoffs between the major programming models for multiprocessors.

3.7 Implications for Programming Models

We have seen in this and the previous chapter that while the decomposition and assignment of a parallel program are often (but not always) independent of the programming model, the orchestration step is highly dependent on it. In Chapter 1, we learned about the fundamental design issues that apply to any layer of the communication architecture, including the programming model. We learned that the two major programming models—a shared address space and explicit message passing between private address spaces—are fundamentally distinguished by functional differences such as naming, replication and synchronization, and that the positions taken on these influence (and are influenced by) performance characteristics such as latency and bandwidth. At that stage, we could only speak about these issues in the abstract, and could not appreciate the interactions with applications and the implications for which programming models are preferable under what circumstances. Now that we have an in-depth understanding of several interesting parallel applications, and we understand the performance issues in orchestration, we are ready to compare the programming models in light of application and performance characteristics.

We will use the application case studies to illustrate the issues. For a shared address space, we assume that loads and stores to shared data are the only communication mechanisms exported to the user, and we call this a load-store shared address space. Of course, in practice there is nothing to stop a system from providing support for explicit messages as well as these primitives in a shared address space model, but we ignore this possibility for now. The shared address space model can be supported in a wide variety of ways at the communication abstraction and hardware-software interface layers (recall the discussion of naming models at the end of Chapter 1) with different granularities and different efficiencies for supporting communication, replication, and coherence. These will be discussed in detail in Chapters 8 and 9. Here we focus on the most common case, in which a cache-coherent shared address space is supported efficiently at fine

granularity; for example, with direct hardware support for a shared physical address space, as well as for communication, replication, and coherence at the fixed granularity of cache blocks. For both the shared address space and message passing models, we assume that the programming model is supported efficiently by the communication abstraction and the hardware-software interface.

As programmers, the programming model is our window to the communication architecture. Differences between programming models and how they are implemented have implications for ease of programming, for the structuring of communication, for performance, and for scalability. The major aspects that distinguish the two programming models are functional aspects like *naming*, *replication* and *synchronization*, organizational aspects like the *granularity* at which communication is performed, and performance aspects such as end-point *overhead* of a communication operation. Other performance aspects such as latency and bandwidth depend largely on the network and network interface used, and can be assumed to be equivalent. In addition, there are differences in *hardware overhead and complexity* required to support the abstractions efficiently, and in the ease with which they allow us to reason about or *predict performance*. Let us examine each of these aspects. The first three point to advantages of a load/store shared address space, while the others favor explicit message passing.

Naming

We have already seen that a shared address space makes the naming of logically shared data much easier for the programmer, since the naming model is similar to that on a uniprocessor. Explicit messages are not necessary, and a process need not name other processes or know which processing node currently owns the data. Having to know or determine which process's address space data reside in and transfer data in explicit messages is not difficult in applications with regular, statically predictable communication needs, such as the equation solver kernel and Ocean. But it can be quite difficult, both algorithmically and for programming, in applications with irregular, unpredictable data needs. An example is Barnes-Hut, in which the parts of the tree that a process needs to compute forces on its bodies are not statically predictable, and the ownership of bodies and tree cells changes with time as the galaxy evolves, so knowing which processes to obtain data from is not easy. Raytrace is another example, in which rays shot by a process bounce unpredictably around scene data that are distributed among processors, so it is difficult to determine who owns the next set of data needed. These difficulties can of course be overcome in both cases, but this requires either changing the algorithm substantially from the uniprocessor version (for example, adding an extra time-step to compute who needs which data and transferring those data to them in Barnes-Hut [Sal90], or using a scene-oriented rather than a ray-oriented approach in Raytrace, as discussed in Section 3.6.3) or emulating an application-specific shared address space in software by hashing bodies, cells or scene data to processing nodes. We will discuss these solutions further in Chapter 7 when we discuss the implications of message passing systems for parallel software.

Replication

Several issues distinguish how replication of nonlocal data is managed on different system: (i) *who* is responsible for doing the replication, i.e. for making local copies of the data? (ii) *where* in the local memory hierarchy is the replication done? (iii) at *what granularity* are data allocated in the replication store? (iv) how are the values of replicated data kept *coherent*? (v) how is the *replacement* of replicated data managed?

With the separate virtual address spaces of the message-passing abstraction, the only way to replicate communicated data is to copy the data into a process's private address space explicitly in the application program. The replicated data are explicitly renamed in the new process's private address space, so the virtual and physical addresses may be different for the two processes and their copies have nothing to do with each other as far as the system is concerned. Organizationally, data are always replicated in main memory first (when the copies are allocated in the address space), and only data from the local main memory enter the processor cache. The granularity of allocation in the local memory is variable, and depends entirely on the user. Ensuring that the values of replicated data are kept up to date (coherent) must be done by the program through explicit messages. We shall discuss replacement shortly.

We mentioned in Chapter 1 that in a shared address space, since nonlocal data are accessed through ordinary processor loads and stores and communication is implicit, there are opportunities for the system to replicate data transparently to the user—without user intervention or explicit renaming of data—just as caches do in uniprocessors. This opens up a wide range of possibilities. For example, in the shared physical address space system we assume, nonlocal data enter the processor's cache subsystem upon reference by a load or store instruction, without being replicated in main memory. Replication happens very close to the processor and at the relatively fine granularity of cache blocks, and data are kept coherent by hardware. Other systems may replicate data transparently in main memory—either at cache block granularity through additional hardware support or at page or object granularity through system software—and may preserve coherence through a variety of methods and granularities that we shall discuss in Chapter 9. Still other systems may choose not to support transparent replication and/or coherence, leaving them to the user.

Finally, let us examine the replacement of locally replicated data due to finite capacity. How replacement is managed has implications for the amount of communicated data that needs to be replicated at a level of the memory hierarchy. For example, hardware caches manage replacement automatically and dynamically with every reference and at a fine spatial granularity, so that the cache needs to be only as large as the active working set of the workload (replication at coarser spatial granularities may cause fragmentation and hence increase the replication capacity needed). When replication is managed by the user program, as in message passing, it is difficult to manage replacement this dynamically. It can of course be done, for example by maintaining a cache data structure in local memory and using it to emulate a hardware cache for communicated data. However, this complicates programming and is expensive: The software cache must be looked up in software on every reference to see if a valid copy already exists there; if it does not, the address must be checked to see whether the datum is locally allocated or a message should be generated.

Typically, message-passing programs manage replacement less dynamically. Local copies of communicated data are allowed to accumulate in local memory, and are flushed out explicitly at certain points in the program, typically when it can be determined that they are not needed for some time. This can require substantially more memory overhead for replication in some applications such as Barnes-Hut and Raytrace. For read-only data, like the scene in Raytrace many message passing programs simply replicate the entire data set on every processing node, solving the naming problem and almost eliminating communication, but also eliminating the ability to run larger problems on larger systems. In the Barnes-Hut application, while one approach emulates a shared address space and also a cache for replication—"flushing" the cache at phase boundaries—in the other approach a process first replicates locally all the data it needs to compute forces on all its assigned bodies, and only then begins to compute forces. While this means there

is no communication during the force calculation phase, the amount of data replicated in main memory is much larger than the process's assigned partition, and certainly much larger than the active working set which is the data needed to compute forces on only one particle (Section 3.6.2). This active working set in a shared address space typically fits in the processor cache, so there is not need for replication in main memory at all. In message passing, the large amount of replication limits the scalability of the approach.

Overhead and Granularity of Communication

The overheads of initiating and receiving communication are greatly influenced by the extent to which the necessary tasks can be performed by hardware rather than being delegated to software, particularly the operating system. As we saw in Chapter 1, in a shared physical address space the underlying uniprocessor hardware mechanisms suffice for address translation and protection, since the shared address space is simply a large flat address space. Simply doing address translation in hardware as opposed to doing it in software for remote references was found to improve Barnes-Hut performance by about 20% in one set of experiments [ScL94]. The other major task is buffer management: incoming and outgoing communications need to be temporarily buffered in the network interface, to allow multiple communications to be in progress simultaneously and to stage data through a communication pipeline. Communication at the fixed granularity of cache blocks makes it easy to manage buffers very efficiently in hardware. These factors combine to keep the overhead of communicating each cache block quite low (a few cycles to a few tens of cycles, depending on the implementation and integration of the communication assist).

In message-passing systems, local references are just loads and stores and thus incur no more overhead than on a uniprocessor. Communication messages, as we know, are much more flexible. They are typically of a variety of possible types (see Section 2.4.6 and the tag matching discussed in Chapter 1), of arbitrary lengths, and between arbitrary address spaces. The variety of types requires software overhead to decode the type of message and execute the corresponding handler routine at the sending or receiving end. The flexible message length, together with the use of asynchronous and nonblocking messages, complicates buffer management, so that the operating system must often be invoked to temporarily store messages. Finally, sending explicit messages between arbitrary address spaces requires that the operating system on a node intervene to provide protection. The software overhead needed to handle buffer management and protection can be substantial, particularly when the operating system must be invoked. A lot of recent design effort has focused on streamlined network interfaces and message-passing mechanisms, which have significantly reduced per-message overheads. These will be discussed in Chapter 7. However, the issues of flexibility and protection remain, and the overheads are likely to remain several times as large as those of load-store shared address space interfaces. Thus, explicit message-passing is not likely to sustain as fine a granularity of communication as a hardware-supported shared address space.

These three issues—naming, replication, and communication overhead—have pointed to the advantages of an efficiently supported shared address space from the perspective of creating a parallel program: The burdens of naming and often replication/coherence are on the system rather than the program, and communication need not be structured in large messages to amortize overhead but can be done naturally through loads and stores. The next four issues, however, indicate advantages of explicit communication.

Block Data Transfer

Implicit communication through loads and stores in a cache-coherent shared address space typically causes a message to be generated per reference that requires communication. The communication is usually initiated by the process that needs the data, and we call it receiver-initiated. Each communication brings in a fixed, usually small amount of data: a cache block in our hardware cache-coherent system. While the hardware support provides efficient fine-grained communication, communicating one cache block at a time is not the most efficient way to communicate a large chunk of data from one processor to another. We would rather amortize the overhead and latency by communicating the data in a single message or a group of large messages. Explicit communication, as in message passing, allows greater flexibility in choosing the sizes of messages and also in choosing whether communication is receiver-initiated or sender-initiated. Explicit communication can be added to a shared address space naming model, and it is also possible for the system to make communication coarser-grained transparently underneath a load-store programming model in some cases of predictable communication. However, the natural communication structure promoted by a shared address space is fine-grained and usually receiver-initiated. The advantages of block transfer are somewhat complicated by the availability of alternative latency tolerance techniques, as we shall see in Chapter 11, but it clearly does have advantages.

Synchronization

The fact that synchronization can be contained in the (explicit) communication itself in message passing, while it is usually explicit and separate from data communication in a shared address space is an advantage of the former model. However, the advantage becomes less significant when asynchronous message passing is used and separate synchronization must be employed to preserve correctness.

Hardware Cost and Design Complexity

The hardware cost and design time required to efficiently support the desirable features of a shared address space above are greater those required to support a message-passing abstraction. Since all transactions on the memory bus must be observed to determine when nonlocal cache misses occur, at least some functionality of the communication assist be integrated quite closely into the processing node. A system with transparent replication and coherence in hardware caches requires further hardware support and the implementation of fairly complex coherence protocols. On the other hand, in the message-passing abstraction the assist does not need to see memory references, and can be less closely integrated on the I/O bus. The actual tradeoffs in cost and complexity for supporting the different abstractions will become clear in Chapters 5, 7 and 8.

Cost and complexity, however, are more complicated issues than simply assist hardware cost and design time. For example, if the amount of replication needed in message-passing programs is indeed larger than that needed in a cache-coherent shared address space (due to differences in how replacement is managed, as discussed earlier, or due to replication of the operating system), then the memory required for this replication should be compared to the hardware cost of supporting a shared address space. The same goes for the recurring cost of developing effective programs on a machine. Cost and price are also determined largely by volume in practice.

Performance Model

In designing parallel programs for an architecture, we would like to have at least a rough performance model that we can use to predict whether one implementation of a program will be better than another and to guide the structure of communication. There are two aspects to a performance model: modeling the cost of primitive events of different types—e.g. communication messages—and modeling the occurrence of these events in a parallel program (e.g. how often they occur, in how bursty a manner, etc.). The former is usually not very difficult, and we have seen a simple model of communication cost in this chapter. The latter can be quite difficult, especially when the programs are complex and irregular, and it is this that distinguishes the performance modeling ease of a shared address space from that of message passing. This aspect is significantly easier when the important events are explicitly identified than when they are not. Thus, the message passing abstraction has a reasonably good performance model, since all communication is explicit. The performance guidelines for a programmer are at least clear: *messages are expensive; send them infrequently.* In a shared address space, particularly a cache-coherent one, performance modeling is complicated by the same property that makes developing a program easier: Naming, replication and coherence are all implicit, so it is difficult to determine how much communication occurs and when. Artifactual communication is also implicit and is particularly difficult to predict (consider cache mapping conflicts that generate communication!), so the resulting programming guideline is much more vague: try to exploit temporal and spatial locality and use data layout when necessary to keep communication levels low. The problem is similar to how implicit caching can make performance difficult to predict even on a uniprocessor, thus complicating the use of the simple *von Neumann* model of a computer which assumes that all memory references have equal cost. However, it is of much greater magnitude here since the cost of communication is much greater than that of local memory access on a uniprocessor.

In summary, the major potential advantages of implicit communication in the shared address space abstraction are programming ease and performance in the presence of fine-grained data sharing. The major potential advantages of explicit communication, as in message passing, are the benefits of block data transfer, the fact that synchronization is subsumed in message passing, the better performance guidelines and prediction ability, and the ease of building machines.

Given these tradeoffs, the questions that an architect has to answer are:

> Is it worthwhile to provide hardware support for a shared address space (i.e. transparent naming), or is it easy enough to manage all communication explicitly?

> If a shared address space is worthwhile, is it also worthwhile to provide hardware support for transparent replication and coherence?

> If the answer to either of the above questions is yes, then is the implicit communication enough or should there also be support for explicit message passing among processing nodes that can be used when desired (this raises a host of questions about how the two abstractions should be integrated), which will be discussed further in Chapter 11.

The answers to these questions depend both on application characteristics and on cost. Affirmative answers to any of the questions naturally lead to other questions regarding how efficiently the feature should be supported, which raise other sets of cost, performance and programming tradeoffs that will become clearer as we proceed through the book. Experience shows that as applications become more complex, the usefulness of transparent naming and replication increases, which argues for supporting a shared address space abstraction. However, with com-

munication naturally being fine-grained, and large granularities of communication and coherence causing performance problems, supporting a shared address space effectively requires an aggressive communication architecture.

3.8 Concluding Remarks

The characteristics of parallel programs have important implications for the design of multiprocessor architectures. Certain key observations about program behavior led to some of the most important advances in uniprocessor computing: The recognition of temporal and spatial locality in program access patterns led to the design of caches, and an analysis of instruction usage led to reduced instruction set computing. In multiprocessors, the performance penalties for mismatches between application requirements and what the architecture provides are much larger, so it is all the more important that we understand the parallel programs and other workloads that are going to run on these machines. This chapter, together with the previous one, has focussed on providing this understanding.

Historically, many different parallel architectural genres led to many different programming styles and very little portability. Today, the architectural convergence has led to common ground for the development of portable software environments and programming languages. The way we think about the parallelization process and the key performance issues is largely similar in both the shared address space and message passing programming models, although the specific granularities, performance characteristics and orchestration techniques are different. While we can analyze the tradeoffs between the shared address space and message passing programming models, both are flourishing in different portions of the architectural design space.

Another effect of architectural convergence has been a clearer articulation of the performance issues against which software must be designed. Historically, the major focus of theoretical parallel algorithm development has been the PRAM model discussed in Section 3.2.4, which ignores data access and communication cost and considers only load balance and extra work (some variants of the PRAM model some serialization when different processors try to access the same data word). This is very useful in understanding the inherent concurrency in an application, which is the first conceptual step; however, it does not take important realities of modern systems into account, such as the fact that data access and communication costs can easily dominate execution time. Historically, communication has been treated separately, and the major focus in its treatment has been mapping the communication to different network topologies. With a clearer understanding of the importance of communication and the important costs in a communication transaction on modern machines, two things have happened. First, models that help analyze communication cost and structure communication have been developed, such as the bulk synchronous programming (BSP) model [Val90] and the LogP model [CKP+93], with the hope of replacing the PRAM as the de facto model used for parallel algorithm analysis. These models strive to expose the important costs associated with a communication event—such as latency, bandwidth, overhead—as we have done in this chapter, allowing an algorithm designer to factor them into the comparative analysis of parallel algorithms. The BSP model also provides a framework that can be used to reason about communication and parallel performance. Second, the emphasis in modeling communication cost has shifted to the cost of communication at the end points, so the number of messages and contention at the end points have become more important than mapping to network topologies. In fact, both the BSP and LogP models ignore network topology completely, modeling network latency as a fixed parameter!

Models such as BSP and LogP are important steps toward a realistic architectural model against which to design and analyze parallel algorithms. By changing the values of the key parameters in these models, we may be able to determine how an algorithm would perform across a range of architectures, and how it might be best structured for different architectures or for portable performance. However, much more difficult than modeling the architecture as a set of parameters is modeling the behavior of the algorithm or application, which is the other side of the modeling equation [SRG94]: What is the communication to computation ratio for irregular applications, how does it change with replication capacity, how do the access patterns interact with the granularities of the extended memory hierarchy, how bursty is the communication and how can this be incorporated into the performance model. Modeling techniques that can capture these characteristics for realistic applications and integrate them into the use of BSP or LogP have yet to be developed.

This chapter has discussed some of the key performance properties of parallel programs and their interactions with basic provisions of a multiprocessor's memory and communications architecture. These properties include load balance; the communication to computation ratio; aspects of orchestrating communication that affect communication cost; data locality and its interactions with replication capacity and the granularities of allocation, transfer and perhaps coherence to generate artifactual communication; and the implications for communication abstractions that a machine must support. We have seen that the performance issues trade off with one another, and that the art of producing a good parallel program is to obtain the right compromise between the conflicting demands. Performance enhancement techniques can take considerable programming effort, depending on both the application and the system, and the extent and manner in which different techniques are incorporated can greatly affect the characteristics of the workload presented to the architecture. Programming for performance is also a process of successive refinement, where decisions made in the early steps may have to be revisited based on system or program characteristics discovered in later steps. We have examined the four application case studies that were introduced in Chapter 2 in depth, and seen how these issues play out in them. We shall encounter several of these performance issues again in more detail as we consider architectural design options, tradeoffs and evaluation in the rest of the book. However, with the knowledge of parallel programs that we have developed, we are now ready to understand how to use the programs as workloads to evaluate parallel architectures and tradeoffs.

3.9 References

[AiN88]   Alexander Aiken and Alexandru Nicolau. Optimal Loop Parallelization. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 308-317, 1988.

[AC+95]   Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg and Katherine Yelick. Empirical Evaluation of the Cray-T3D: A Compiler Perspective. Proceedings of the Twenty-second International Symposium on Computer Architecture, pp. 320-331, June 1995.

[Bu+92]   Burkhardt, H. et al, Overview of the KSR-1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[CM88]   Chandy, K.M. and Misra, J. Parallel Program Design: A Foundation. Addison Wesley, 1988.

[CKP+93]   David Culler et al. LogP: Toward a realistic model of parallel computation. In *Proceedings of the Principles and Practice of Parallel Processing*, pages 1-12, 1993.

[Da+87]   Dally, W.J. et al. The J-Machine: A Fine-grained Concurrent Computer. Proceedings of the IFIPS

World Computer Congress, pp. 1147-1153, 1989.

[Den68]   Denning, P.J. The Working Set Model for Program Behavior. Communications of the ACM, vol. 11, no. 5, May 1968, pp. 323-333.

[DS68]   Dijkstra, E.W. and Sholten, C.S. Termination Detection for Diffusing Computations. Information Processing Letters 1, pp. 1-4.

[FoW78]   S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In Proceedings of the Tenth ACM Symposium on Theory of Computing, May 1978.

[GP90]   Green, S.A. and Paddon, D.J. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, vol. 6, 1990, pp. 62-73.

[Hil85]   Hillis, W.D. The Connection Machine. MIT Press, 1985.

[HiS86]   Hillis, W.D. and Steele, G.L. Data Parallel Algorithms. Communications of the ACM, 29(12), pp. 1170-1183, December 1986.

[HoC92]   Hockney, R. W., and <<xxx>>. Comparison of Communications on the Intel iPSC/860 and Touchstone Delta. Parallel Computing, 18, pp. 1067-1072, 1992.

[KG+94]   V. Kumar, A.Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Lei+92]   Charles E. Leiserson, Z.S. Abuhamdeh, D. C. Douglas, C.R. Feynmann, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M. St. Pierre, D.S. Wells, M.C. Wong, S-W Yang and R. Zak. The Network Architecture of the Connection Machine CM-5. In Proceedings of the Symposium on Parallel Algorithms and Architectures, pp. 272-285, June 1992.

[LLJ+93]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[LO+87]   Lusk, E.W., Overbeek, R. et al. Portable Programs for Parallel Processors. Holt, Rinehart and Winston, Inc. 1987.

[NWD93]   Noakes, M.D., Wallach, D.A., and Dally, W.J. The J-Machine Multicomputer: An Architectural Evaluation. Proceedings of the Twentieth International Symposium on Computer Architecture, May 1993, pp. 224-235.

[Pie88]   Pierce, Paul. The NX/2 Operating System. Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pp. 384--390, January, 1988.

[Sal90]   Salmon, J. Parallel Hierarchical N-body Methods. Ph.D. Thesis, California Institute of Technology, 1990.

[WaS93]   Warren, Michael S. and Salmon, John K. A Parallel Hashed Oct-Tree N-Body Algorithm. Proceedings of Supercomputing'93, IEEE Computer Society, pp. 12-21, 1993.

[SWW94]   Salmon, J.K., Warren, M.S. and Winckelmans, G.S. Fast parallel treecodes for gravitational and fluid dynamical N-body problems. Intl. Journal of Supercomputer Applications, vol. 8, pp. 129 - 142, 1994.

[ScL94]   D. J. Scales and M. S. Lam. Proceedings of the First Symposium on Operating System Design and Implementation, November, 1994.

[SGL94]   Singh, J.P., Gupta, A. and Levoy, M. Parallel Visualization Algorithms: Performance and Architectural Implications. IEEE Computer, vol. 27, no. 6, June 1994.

[SH+95]   Singh, J.P., Holt, C., Totsuka, T., Gupta, A. and Hennessy, J.L. Load balancing and data locality in Hierarchial N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. Journal of Parallel and Distributed Computing, 1995 <<complete citation>>.

[SJG+93]   Singh, J.P., Joe T., Gupta, A. and Hennessy, J. An Empirical Comparison of the KSR-1 and DASH Multiprocessors. Proceedings of Supercomputing'93, November 1993.

[SRG94]   Jaswinder Pal Singh, Edward Rothberg and Anoop Gupta. Modeling Communication in Parallel Algorithms: A Fruitful Interaction Between Theory and Systems? In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[SWG91]   Singh, J.P., Weber, W-D., and Gupta, A. SPLASH: The Stanford ParalleL Applications for Shared Memory. Computer Architecture News, vol. 20, no. 1, pp. 5-44, March 1992.

[Val90]   Leslie G. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM, vol. 33, no. 8, August 1990, pp. 103-111.

3.10   Exercises

3.1 Short Answer Questions:

    a. For which of the applications that we have described (Ocean, Galaxy, Raytrace) can we be described to have followed this view of decomposing data rather than computation and using an owner-computes rule in our parallelization? What would be the problem(s) with using a strict data distribution and owner-computes rule in the others? How would you address the problem(s)?

    b. What are the advantages and disadvantages of using distributed task queues (as opposed to a global task queue) to implement load balancing?

    c. Do small tasks inherently increase communication, contention and task management overhead?

    d. Draw *one* arc from each kind of memory system traffic below (the list on the left) to the solution technique (on the right) that is *the most effective way* to reduce that source of traffic in a machine that supports a shared address space with physically distributed memory.

Kinds of Memory System Traffic	Solution Techniques
Cold-Start Traffic	Large Cache Sizes
Inherent Communication	Data Placement
Extra Data Communication on a Miss	Algorithm Reorganization
Capacity-Generated Communication	Larger Cache Block Size
Capacity-Generated Local Traffic	Data Structure Reorganization

    e. Under what conditions would the sum of busy-useful time across processes (in the execution time) not equal the busy-useful time for the sequential program, assuming both the sequential and parallel programs are deterministic? Provide examples.

3.2 Levels of parallelism.

    a. As an example of hierarchical parallelism, consider an algorithm frequently used in medical diagnosis and economic forecasting. The algorithm propagates information through a network or graph, such as the one in Figure 3-22. Every node represents a matrix of values. The arcs correspond to dependences between nodes, and are the channels along which information must flow. The algorithm starts from the nodes at the bottom of the graph and works upward, performing matrix operations at every node encountered along the way. It affords parallelism at least two levels: Nodes that do not have an ancestor-descendent relationship in the traversal can be computed in parallel, and the matrix computations within a node can be parallelized as well. How would you parallelize this algorithm? What are the tradeoffs that are most important?
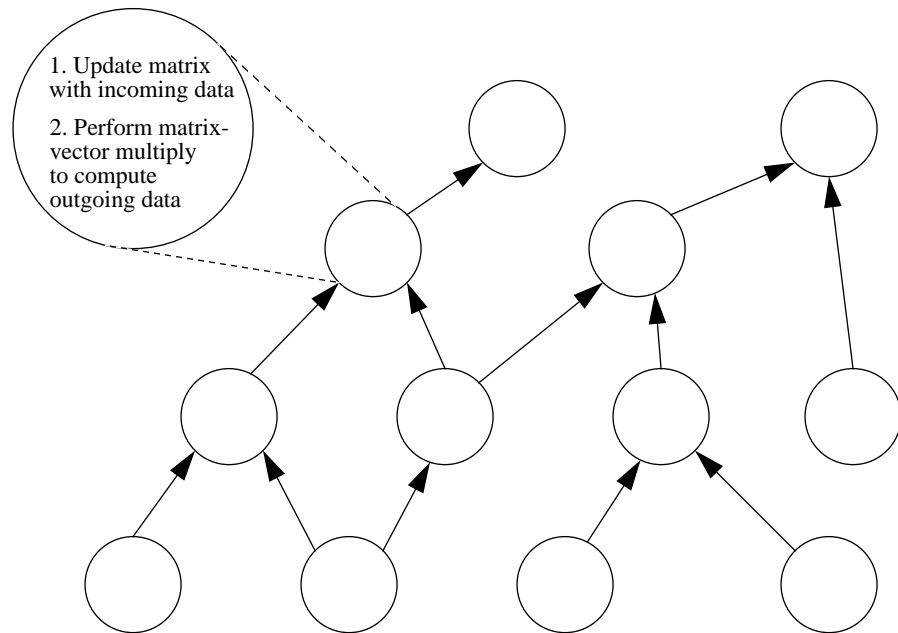
Figure 3-22 Levels of parallelism in a graph computation.

The work within a graph node is shown in the expanded node o the left.

    b. Levels of parallelism in locusroute. What are the tradeoffs in determining which level to pick. What parameters affect your decision? What you would you pick for this case: 30 wires, 24 processors, 5 segments per wire, 10 routes per segment, each route evaluation takes same amount of time? If you had to pick one and be tied to it for all cases, which would you pick (you can make and state reasonable assumptions to guide your answer)?

3.3 LU factorization.

    a. Analyze the load balance and communication volume for both cases of broadcast based LU with row decomposition: block assignment and interleaved assignment.

    b. What if we used a two-dimensional scatter decomposition at the granularity of individual elements. Analyze the load balance and communication volume.

    c. Discuss the tradeoffs (programming difficulty and likely performance differences) in programming the different versions in the different communication abstractions. Would you expect one to always be better?

    d. Can you think of a better decomposition and assignment? Discuss.

    e. Analyze the load balance and communication volume for a pipelined implementation in a 2-d scatter decomposition of elements.

    f. Discuss the tradeoffs in the performance of the broadcast versus pipelined versions.

3.4 If E is the set of sections of the algorithm that are enhanced through parallelism, f_k is the fraction of the sequential execution time taken up by the kth enhanced section when run on a uniprocessor, and s_k is the speedup obtained through parallelism on the kth enhanced section, derive an expression for the overall speedup obtained. Apply it to the broadcast approach for

LU factorization at element granularity. Draw a rough concurrency profile for the computation (a graph showing the amount of concurrency versus time, where the unit of time is a logical operation, say updating an interior active element, performed on one or more processors. Assume a 100-by-100 element matrix. Estimate the speedup, ignoring memory referencing and communication costs.

3.5 We have discussed the technique of blocking that is widely used in linear algebra algorithms to exploit temporal locality (see Section 3.4.1). Consider a sequential LU factorization program as described in this chapter. Compute the read miss rate on a system with a cache size of 16KB and a matrix size of 1024-by-1024. Ignore cache conflicts, and count only access to matrix elements. Now imagine that LU factorization is blocked for temporal locality, and compute the miss rate for a sequential implementation (this may be best done after reading the section describing the LU factorization program in the next chapter). Assume a block size of 32-by-32 elements, and that the update to a B-by-B block takes B^3 operations and B^2 cache misses. Assume no reuse of blocks across block operations. If read misses cost 50 cycles, what is the performance difference between the two versions (counting operations as defined above and ignoring write accesses).

3.6 It was mentioned in the chapter that termination detection is an interesting aspect of task stealing. Consider a task stealing scenario in which processes produce tasks as the computation is ongoing. Design a good tasking method (where to take tasks from, how to put tasks into the pool, etc.) and think of some good termination detection heuristics. Perform worst-case complexity analysis for the number of messages needed by the termination detection methods you consider. Which one would you use in practice? Write pseudocode for one that is guaranteed to work and should yield good performance

3.7 Consider transposing a matrix in parallel, from a source matrix to a destination matrix, i.e. B[i,j] = A[j,i].

 a. How might you partition the two matrices among processes? Discuss some possibilities and the tradeoffs. Does it matter whether you are programming a shared address space or message passing machine?

 b. Why is the interprocess communication in a matrix transpose called all-to-all personalized communication?

 c. Write simple pseudocode for the parallel matrix transposition in a shared address space and in message passing (just the loops that implement the transpose). What are the major performance issues you consider in each case, other than inherent communication and load balance, and how do you address them?

 d. Is there any benefit to blocking the parallel matrix transpose? Under what conditions, and how would you block it (no need to write out the full code)? What, if anything, is the difference between blocking here and in LU factorization?

3.8 The communication needs of applications, even expressed in terms of bytes per instruction, can help us do back-of-the-envelope calculations to determine the impact of increased bandwidth or reduced latency. For example, a Fast Fourier Transform (FFT) is an algorithm that is widely used in digital signal processing and climate modeling applications. A simple parallel FFT on n data points has a per-process computation cost of $O(n \log n / p)$, and per-process communication volume of $O(n/p)$, where p is the number of processes. The communication to computation ratio is therefore $O(1/\log n)$. Suppose for simplicity that all the constants in the above expressions are unity, and that we are performing an n=1M (or 2^{20}) point FFT on p=1024 processes. Let the average communication latency for a word of data (a point in the FFT) be 200 processor cycles, and let the communication bandwidth between any node and the network be 100MB/sec. Assume no load imbalance or synchronization costs.

    a. With no latency hidden, for what fraction of the execution time is a process stalled due to communication latency?

    b. What would be the impact on execution time of halving the communication latency?

    c. What are the node-to-network bandwidth requirements without latency hiding?

    d. What are the node-to-network bandwidth requirements assuming all latency is hidden, and does the machine satisfy them?

3.9 What kind of data (local, nonlocal, or both) constitute the relevant working set in: (i) main memory in a message-passing abstraction, (ii) processor cache in a message-passing abstraction, (iii) processor cache in cache-coherent shared address space abstraction.

3.10 Implement a reduction and a broadcast. First an O(p) linear method, then an O(log p) i.e. tree based, method. Do this both for a shared address space and for message-passing.

3.11 After the assignment of tasks to processors, there is still the issue of scheduling the tasks that a process is assigned to run in some temporal order. What are the major issues involved here? Which are the same as on uniprocessor programs, and which are different? Construct examples that highlight the impact of poor scheduling in the different cases.

3.12 In the data mining case study, why are 2-itemsets computed from the original format of the database rather than the transformed format?

3.13 You have been given the job of creating a word count program for a major book publisher. You will be working on a shared-memory multiprocessor with 32 processing elements. Your only stated interface is "get_words" which returns an array of the book's next 1000 words to be counted. The main work each processor will do should look like this:

```
while(get_words(word)) {
    for (i=0; i<1000; i++) {
        if word[i] is in list
            increment its count
        else
            add word to list
    }
}
/* Once all words have been logged,
the list should printed out */
```

Using pseudocode, create a detailed description of the control flow and data structures that you will use for this program. Your method should attempt to minimize space, synchronization overhead, and memory latency. This problem allows you a lot of flexibility, so state all assumptions and design decisions.

CHAPTER 4   # Workload-Driven Evaluation

4.1 Introduction

The field of computer architecture is becoming increasingly quantitative. Design features are adopted only after detailed evaluations of tradeoffs. Once systems are built, they are evaluated and compared both by architects to understand their tradeoffs and by users to make procurement decisions. In uniprocessor design, with a rich base of existing machines and widely used applications, the process of evaluating tradeoffs is one of careful extrapolation from known quantities. Isolated performance characteristics of the machines can be obtained from microbenchmarks, small programs that stress a particular machine feature. Popular workloads are codified in standard benchmarks suites, such as the Standard Performance Evaluation Corporation (SPEC) benchmark suite [SPE89] for engineering workloads, and measurements are made on a range of existing design alternatives. Based on these measurements, assessments of emerging technology, and expected changes in requirements, designers propose new alternatives. The ones that appear promising are typically evaluated through simulation. First, a *simulator*, a program that simulates

the design with and without the proposed feature of interest, is written. Then a number of programs, or multiprogrammed workloads, representative of those that are likely to run on the machine are chosen. These workloads are run through the simulator and the performance impact of the feature determined. This, together with the estimated cost of the feature in hardware and design time, determines whether the feature will be included. Simulators are written to be flexible, so organizational and performance parameters can be varied to understand their impact as well.

Good workload-driven evaluation is a difficult and time consuming process, even for uniprocessor systems. The workloads need to be renewed as technology and usage patterns change. Industry-standard benchmark suites are revised every few years. In particular, the input data sets used for the programs affect many of the key interactions with the systems, and determine whether or not the important features of the system are stressed. For example, to take into account the huge increases in processor speeds and changes in cache sizes, a major change from SPEC92 to SPEC95 was the use of larger input data sets to stress the memory system. Accurate simulators are costly to develop and verify, and the simulation runs consume huge amounts of computing time. However, these efforts are well rewarded, because good evaluation yields good design.

As multiprocessor architecture has matured and greater continuity has been established from one generation of machines to the next, a similar quantitative approach has been adopted. Whereas early parallel machines were in many cases like bold works of art, relying heavily on the designer's intuition, modern design involves considerable evaluation of proposed design features. Here too, workloads are used to evaluate real machines as well as to extrapolate to proposed designs and explore tradeoffs through software simulation. For multiprocessors, the workloads of interest are either parallel programs or multiprogrammed mixes of sequential and parallel programs. Evaluation is a critical part of the new, engineering approach to multiprocessor architecture; it is very important that we understand the key issues in it before we go forward into the core of multiprocessor architecture and our own evaluation of tradeoffs in this book.

Unfortunately, the job of workload-driven evaluation for multiprocessor architecture is even more difficult than for uniprocessors, for several reasons:

Immaturity of parallel applications It is not easy to obtain "representative" workloads for multiprocessors, both because their use is relatively immature and because several new behavioral axes arise to be exercised.

Immaturity of parallel programming languages The software model for parallel programming has not stabilized, and programs written assuming different models can have very different behavior.

Sensitivity of behavioral differences Different workloads, and even different decisions made in parallelizing the same sequential workload, can present vastly different execution characteristics to the architecture.

New degrees of freedom There are several new degrees of freedom in the architecture. The most obvious is the number of processors. Others include the organizational and performance parameters of the extended memory hierarchy, particularly the communication architecture. Together with the degrees of freedom of the workload (i.e. application parameters) and the underlying uniprocessor node, these parameters lead to a very large design space for experimentation, particularly when evaluating an idea or tradeoff in a general context rather than evaluating a fixed machine. The high cost of communication makes performance much more

sensitive to interactions among all these degrees of freedom than it is in uniprocessors, making it all the more important that we understand how to navigate the large parameter space.

Limitations of simulation Simulating multiprocessors in software to evaluate design decisions is more resource intensive than simulating uniprocessors. Multiprocessor simulations consume a lot of memory and time, and unfortunately do not speed up very well when parallelized themselves. Thus, although the design space we wish to explore is larger, the space that we can actually explore is often much smaller. We have to make careful tradeoffs in deciding which parts of the space to simulate.

Our understanding of parallel programs from Chapters 2 and 3 will be critical in dealing with these issues. Throughout this chapter, we will learn that effective evaluation requires understanding the important properties of both workloads and architectures, as well as how these properties interact. In particular, the relationships among application parameters and the number of processors determine fundamental program properties such as communication to computation ratio, load balance, and temporal and spatial locality. These properties interact with parameters of the extended memory hierarchy to influence performance, in application-dependent and often dramatic ways (see Figure 4-1). Given a workload, choosing parameter values and understanding

(a) Ocean on Origin2000　　　　(b) Barnes-Hut on two machines
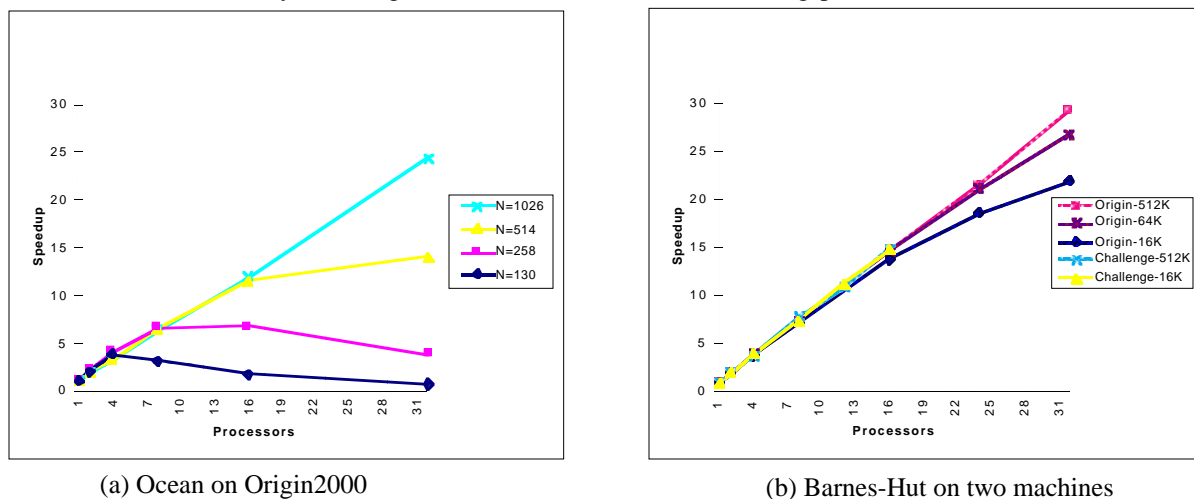
Figure 4-1 Impact of problem size on parallel performance.

For many applications (like Ocean on the left), the effect is dramatic, at least until the problem size becomes large enough for the number of processors. For others, like the Barnes-Hut galaxy simulation on the right, the effect is much smaller for interesting problem sizes.

their scaling relationships and interactions is a crucial aspect of workload-driven evaluation, with far-reaching implications for evaluating both real machines and architectural tradeoffs. It affects the experiments we design for adequate coverage as well as the conclusions of our evaluations, and helps us restrict the number of experiments or parameter combinations we must examine.

An important goal of this chapter is to highlight the key interactions of workload properties and these parameters or *sizes*, illustrate their significance and point out the important pitfalls. While there is no universal formula for evaluation, the chapter articulates a methodology for both evaluating real machines and assessing tradeoffs through simulation This methodology is followed in several illustrative evaluations throughout the book. It is important that we not only perform good

evaluations ourselves, but also understand the limitations of our own and other people's studies so we can keep them in perspective as we make architectural decisions.

The chapter begins by discussing the fundamental issue of scaling workload parameters as the number of processors or size of the system increases, and the implications for performance metrics and the behavioral characteristics of programs. The interactions with parameters of the extended memory hierarchy and how they should be incorporated into the actual design of experiments will be discussed in the next two sections, which examine the two major types of evaluations.

Section 4.3 outlines a methodology for evaluating a real machine. This involves first understanding the types of benchmarks we might use and their roles in such evaluation—including microbenchmarks, kernels, applications, and multiprogrammed workloads—as well as desirable criteria for choosing them. Then, given a workload we examine how to choose its parameters to evaluate a machine, illustrating the important considerations and providing examples of how inappropriate or limited choices may lead us to incorrect conclusions. The section ends with a summary of the roles of various metrics that we might use to interpret and present results.

Section 4.4 extends this methodological discussion to the more challenging problem of evaluating an architectural tradeoff in a more general context through simulation. Having understood how to perform workload-driven evaluation, in Section 4.5 provide the methodologically relevant characteristics of the workloads used in the illustrative evaluations presented in the book. Some important publicly available workload suites for parallel computing and their philosophies are described in APPENDIX A.

4.2 Scaling Workloads and Machines

Suppose we have chosen a parallel program as a workload, and we want to use it to evaluate a machine. For a parallel machine, there are two things we might want to measure: the *absolute performance*, and the *performance improvement due to parallelism*. The latter is typically measured as the speedup, which was defined in Chapter 1 (Section 1.2.1) as the absolute performance achieved on p processors divided by that achieved on a single processor. Absolute performance (together with cost) is most important to the end user or buyer of a machine. However, in itself it does not tell us much about how much of the performance comes from the use of parallelism and the effectiveness of the communication architecture rather than the performance of an underlying single-processor node. Speedup tells us this, but with the caveat that it is easier to obtain good speedup when the individual nodes have lower performance. Both are important, and both should be measured.

Absolute performance is best measured as work done per unit of time. The amount of work to be done is usually defined by the input configuration on which the program operates. This input configuration may either be available to the program up front, or it may specify a set of continuously arriving inputs to a "server" application, such as a system that processes a bank's transactions or responds to inputs from sensors. For example, suppose the input configuration, and hence work, is kept fixed across the set of experiments we perform. We can then treat the work as a fixed point of reference and define performance as the reciprocal of execution time. In some application domains, users find it more convenient to have an explicit representation of work, and use a work-per-unit-time metric even when the input configuration is fixed; for example in a transac-

tion processing system it could be the number of transactions serviced per minute, in a sorting application the number of keys sorted per second, and in a chemistry application the number of bonds computed per second. It is important to realize that even though work is explicitly represented performance is measured with reference to a particular input configuration or amount of work, and that these metrics are derived from measurements of execution time (together with the number of application events of interest). Given a fixed problem configuration, there is no fundamental advantage to these metrics over execution time. In fact, we must be careful to ensure that the measure of work being used is indeed a meaningful measure from the application perspective, not something that one can cheat against. We shall discuss desirable properties of work metrics further as we go along, and return to some of the more detailed issues in metrics in Section 4.3.5. For now, let us focus on evaluating the improvement in absolute performance due to parallelism, i.e. due to using p processors rather than one.

With execution time as our measure of performance, we saw in Chapter 1 that we could simply run the program with the same input configuration on one and p processors, and measure the improvement or speedup as $\frac{Time(1\,proc)}{Time(p\,procs)}$. With operations per second, we can measure speedup as $\frac{OperationsPerSecond(p\,procs)}{OperationsPerSecond(1\,proc)}$. There is a question about how we should measure performance on one processor; for example, it is more honest to use the performance of the best sequential program running on one processor rather than the parallel program itself running on one processor, a point we shall return to in Section 4.3.5. But this is quite easily addressed. As the number of processors is changed, we can simply run the problem on the different numbers of processors and compute speedups accordingly. Why then all the fuss about scaling?

4.2.1 Why Worry about Scaling?

Unfortunately, there are several reasons why speedup measured on a fixed problem is limited as the *only* way of evaluating the performance improvement due to parallelism across a range of machine scales.

Suppose the fixed problem size we have chosen is relatively small, appropriate for evaluating a machine with few processors and exercising their communication architectures. As we increase the number of processors for the same problem size, the overheads due to parallelism (communication, load imbalance) increase relative to useful computation. There will come a point when the problem size is unrealistically small to evaluate the machine at hand. The high overheads will lead to uninterestingly small speedups, which reflect not so much the capabilities of the machine as the fact that an inappropriate problem size was used (say one that does not have enough concurrency for the large machine). In fact, there will come a point when using more processors does not improve performance, and may even hurt it as the overheads begin to dominate useful work (see Figure 4-2(a)). A user would not run this problem on a machine that large, so it is not appropriate for evaluating this machine. The same is true if the problem takes a very small amount of time on the large machine.

On the other hand, suppose we choose a problem that is realistically large for a machine with many processors. Now we might have the opposite problem in evaluating the performance improvement due to parallelism. This problem may be too big for a single processor, in that it may have data requirements that are far too large to fit in the memory of a single node. On some machines it may not be runnable on a single processor; on others the uniprocessor execution will

(a) Small Ocean problem

(b) Superlinear speedup with big equation solver problem
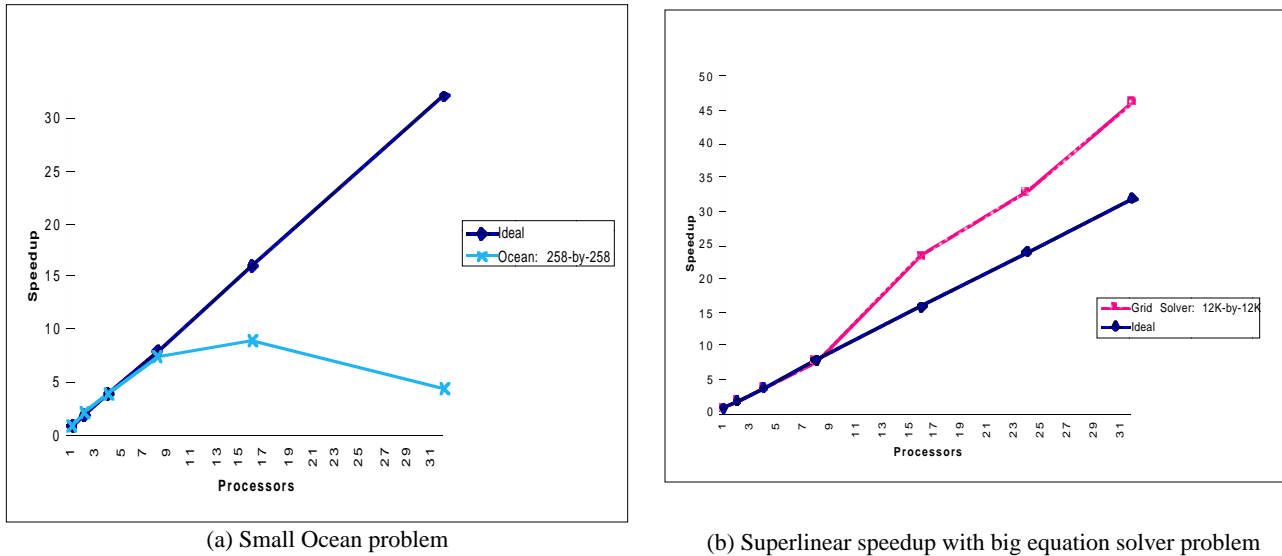
Figure 4-2 Speedups on the SGI Origin2000 as the number of processors increases.

(a) shows the speedup for a small problem size in the Ocean application. The problem size is clearly very appropriate for a machine with 8 or so processors. At a little beyond 16 processors, the speedup has saturated and it is no longer clear that one would run this problem size on this large a machine. And this is clearly the wrong problem size to run on or evaluate a machine with 32 or more processors, since we'd be better off running it on a machine with 8 or 16 processors! (b) shows the speedup for the equation solver kernel illustrating superlinear speedups when a processor's working set fits in cache by the time 16 processors are used, but does not fit and performs poorly when fewer processors are used.

thrash severely to disk; and on still others the overflow data will be allocated in other node's memories in the extended hierarchy, leading to a lot of artifactual communication and hence poor performance. When enough processors are used, the data will fit in their collective memories, eliminating this artifactual communication if the data are distributed properly. The usual speedup is augmented by the great reduction in this artifactual communication, and the result is a speedup far beyond the number of processors used. Once this has happened, the increase in speedup will behave in a more usual way as the number of processors is increased, but the overall speedup over a uniprocessor is still *superlinear* in the number of processors. This situation holds for any level of the memory hierarchy, not just main memory; for example, the aggregate cache capacity of the machine grows as processors are added. An example using cache capacity is illustrated for the equation solver kernel in Figure 4-2(b). This greatly superlinear speedup due to memory system effects is not fake. Indeed, from a user's perspective the availability of more, distributed memory is an important advantage of parallel systems over uniprocessor workstations, since it enables them to run much larger problems and to run them much faster. However, the superlinear speedup does not allow us to separate out the two effects, and as such does not help us evaluate the effectiveness of the machine's communication architecture.

Finally, using the same problem size (input configuration) as the number of processors is changed often does not reflect realistic usage of the machine. Users often want to use more powerful machines to solve larger problems rather than to solve the same problem faster. In these cases, since problem size is changed together with machine size in practical use of the machines, it should be changed when evaluating the machines as well. Clearly, this has the potential to overcome the above problems with evaluating the benefits of parallelism. Together with an apprecia-

tion for how technology scales (for example, processor speeds relative to memory and network speeds), understanding scaling and its implications is also very important for designing next-generation machines and determining appropriate resource distributions for them [RSG93].

We need well-defined scaling models for how problem size should be changed with machine size, so that we can evaluate machines against these models. The measure of performance is always work per unit of time. However, if the problem size is scaled the work done does not stay constant, so we cannot simply compare execution times to determine speedup. Work must be represented and measured, and the question is how. We will discuss these issues in this section, and illustrate the implications of the scaling models for program characteristics such as the communication to computation ratio, load balance, and data locality in the extended memory hierarchy. For simplicity, we focus on the case where the workload is a single parallel application, not a multiprogrammed load. But first, we need to clearly define two terms that we have used informally so far: *scaling a machine*, and *problem size*.

Scaling a Machine means making it more (or less) powerful. This can be done by making any component of the machine bigger, more sophisticated, or faster: the individual processors, the caches, the memory, the communication architecture or the I/O system. Since our interest is in parallelism, we define machine size as the number of processors. We assume that the individual node, its local memory system, and the per-node communication capabilities remain the same as we scale; scaling a machine up thus means adding more nodes, each identical to the ones already present in the machine. For example, scaling a machine with p processors and $p*m$ megabytes of total memory by a factor of k results in a machine with $k*p$ processors and $k*p*m$ megabytes of total memory.

Problem size refers to a specific problem instance or input configuration. It is usually specified by a vector of input parameters, not just a single parameter n (an n-by-n grid in Ocean or n particles in Barnes-Hut). For example, in Ocean, the problem size is specified by a vector $V = (n, \varepsilon, \Delta t, T)$, where n is the grid size in each dimension (which specifies the spatial resolution of our representation of the ocean), ε is the error tolerance used to determine convergence of the multigrid equation solver, Δt is the temporal resolution, i.e. the physical time between time-steps, and T is the number of time-steps performed. In a transaction processing system, it is specified by the number of terminals used, the rate at which users at the terminals issue transactions, the mix of transactions, etc. Problem size is a major factor that determines the work done by the program.

Problem size should be distinguished from *data set size*. The data set size for a parallel program is the amount of storage that would be needed to run the program, assuming no replication; i.e. the amount of main memory that would be needed to fit the problem on a single processor. This is itself distinct from the actual *memory usage* of the program, which is the amount of memory used by the parallel program including replication. The data set size typically depends on a small number of program parameters, most notably the one above that is usually represented as n. In Ocean, for example, the data set size is determined solely by n. The number of instructions and the execution time, however, are not, and depend on all the other problem size parameters as well. The problem size vector will also cause the number of instructions issued (and the execution time) to change. Thus, while the problem size vector V determines many important properties of the application program—such as its data set size, the number of instructions it executes and its execution time—it is not identical to any one of these.

4.2.2 Key Issues in Scaling

Given these definitions, there are two major questions to address when scaling a problem to run on a larger machine:

Under what constraints should the problem be scaled? To define a scaling model, some property must be kept fixed as the machine scales. For example, we may scale so the data set size per processor remains fixed, or the execution time, or the number of transactions executed per second, or the number of particles or rows of a matrix assigned to each processor.

How should the problem be scaled? That is, how should the parameters in the problem-size vector *V* be changed to meet the chosen constraints?

To simplify the discussion, we shall begin by pretending that the problem size is determined by a single parameter *n*, and examine scaling models and their impact under this assumption. Later, in Section 4.2.4, we shall examine the more subtle issue of scaling workload parameters relative to one another.

4.2.3 Scaling Models

Many aspects of a problem and its execution may be used as the basis for scaling constraints. We can divide them into two categories: *user-oriented properties*, and *resource-oriented properties*. Examples of user-oriented properties that we might keep fixed as we scale are the number of particles per processor, the number of rows of a matrix per processor, the number of transactions issued to the system per processor, or the number of I/O operations performed per processor. Examples of resource-oriented constraints are execution time and the total amount of memory used per processor. Each of these defines a distinct scaling model, since the amount of work done for a given number of processors is different when scaling is performed under different constraints. Whether user- or resource-oriented constraints are more appropriate, and which resource- or user-oriented constraint is most useful, depends on the application domain. Our job, or the job of people constructing benchmarks, is to ensure that the scaling constraints we use are meaningful for the domain at hand. User-oriented constraints are usually much easier to follow (e.g. simply change the number of particles linearly with the number of processors). However, large-scale programs are in practice often run under tight resource constraints, and resource constraints are more universal across application domains (time is time and memory is memory, regardless of whether the program deals with particles or matrices). We will therefore use resource constraints to illustrate the effects of scaling models. Let us examine the three most popular resource-oriented models for the constraints under which an application should be scaled to run on a *k* times larger machine:

Problem constrained (PC) scaling Here, the problem size is kept fixed, i.e. is not scaled at all, despite the concerns with a fixed problem size discussed earlier. The same input configuration is used regardless of the number of processors on the machine.

Time constrained (TC) scaling Here, the wall-clock execution time needed to complete the program is held fixed. The problem is scaled so that the new problem's execution time on the large machine is the same as the old problem's execution time on the small machine [Gus88].

Memory constrained (MC) scaling Here, the amount of main memory used per processor is held fixed. The problem is scaled so that the new problem uses exactly *k* times as much main memory (including data replication) as the old problem. Thus, if the old problem just fit in the

memory of the small machine, then the new problem will just fit in the memory of the large machine.

More specialized models are more appropriate in some domains. For example, in its commercial online transaction processing benchmark, the Transaction Processing Council (TPC, discussed later) dictates a scaling rule in which the number of user terminals that generate transactions and the size of the database are scaled proportionally with the "computing power" of the system being evaluated, measured in a specific way. In this as well as in TC and MC scaling, scaling to meet resource constraints often requires some experimentation to find the appropriate input, since resource usage doesn't usually scale cleanly with input parameters. Memory usage is sometimes more predictable—especially if there is no need for replication in main memory—but it is difficult to predict the input configuration that would take the same execution time on 256 processors that another input configuration took on 16. Let us look at each of PC, TC and MC scaling a little further, and see what "work per unit of time" translates to under them.

Problem Constrained Scaling

The assumption in PC scaling is that a user wants to use the larger machine to solve the same problem faster, rather than to solve a larger problem. This is not an unusual situation. For example, if a video compression algorithm currently handles only one frame per second, our goal in using parallelism may not be to compress a larger image in one second, but rather to compress 30 frames per second and hence achieve real-time compression for that frame size. As another example, if a VLSI routing tool takes a week to route a complex chip, we may be more interested in using additional parallelism to reduce the routing time rather than to route a larger chip. However, as we saw in Figure 4-2, the PC scaling model in general makes it difficult to keep achieving good speedups on larger machines. Since work remains fixed, we can use the formulations of the speedup metric that we discussed earlier:

$$Speedup_{PC}(p \text{ processors}) = \frac{Time(1 \text{ processors})}{Time(p \text{ processor})}.$$

Time constrained scaling

This model assumes that users have a certain amount of time that they can wait for a program to execute regardless of the scale of machine being used, and they want to solve the largest possible problem in that fixed amount of time (think of a user who can afford to buy eight hours of computer time at a computer center, or one who is willing to wait overnight for a run to complete but needs to have the results ready to analyze the next morning). While in PC scaling the problem size is kept fixed and the execution time varies, in TC scaling the problem size increases but the execution time is kept fixed. Since performance is work divided by time, and since time stays fixed as the system is scaled, speedup can be measured as the increase in the amount of work done in that execution time:

$$Speedup_{TC}(p \text{ processors}) = \frac{Work(p \text{ processors})}{Work(1 \text{ processor})} \quad ,$$

The question is how to measure work. If we measure it as actual execution time on a single processor, then we would have to run the larger (scaled) problem size on a single processor of the machine to obtain the numerator. But this is fraught with the same thrashing difficulty as before, either in the caches or in the main memory or both, and may frequently lead to superlinear speedups. The desirable properties for a work metric are:

Ease of measurement and architectural independence The metric should be easy to measure and as architecture-independent as possible. Ideally, it should be easily modeled with an analytical expression based only on the application, and we should not have to perform any additional experiments to measure the work in the scaled-up problem.

Linear scaling The measure of work should scale linearly with sequential time complexity of the algorithm.

Example 4-1 Why is the linear scaling property important for a work metric? Illustrate with an example.

Answer The linear scaling property is important if we want the ideal speedup to be linear in the number of processors (ignoring computational non-determinism and memory system artifacts that can make the speedup superlinear). To see this, suppose we use as our work metric the number of rows n in the square matrices that we multiply in matrix multiplication. Let us ignore memory system artifacts. If the uniprocessor problem has n_0 rows, then its execution "time" or the number of multiplication operations it needs to execute will be proportional to n_0^3. This is the problem size. Since the problem is deterministic, ignoring memory system artifacts the best we can hope p processors to do in the same time is $n_0^3 \times p$ operations, which corresponds to $\langle n_0 \times \sqrt[3]{p} \rangle$-by-$\langle n_0 \times \sqrt[3]{p} \rangle$ matrices, even if we assume no load imbalance and no communication cost. If we measure work as the number of rows, then the speedup even in this idealized case will be $\sqrt[3]{p}$ instead of p. Using the number of points in the matrix (n^2) as the work metric also does not work from this perspective, since it would result in an ideal time-constrained speedup of $p^{2/3}$.

The ideal case for measuring work is a measure that satisfies both the above conditions, and is an intuitive parameter from a user's perspective. For example, in sorting integer keys using a method called radix sorting (discussed further in Section 4.5.1), the sequential complexity grows linearly with the number of keys to be sorted, so we can use keys/second as the measure of performance. However, such a measure is difficult to find in real applications, particularly when multiple application parameters are scaled and affect execution time in different ways (see Section 4.2.4). So what should we do?

If we can indeed find a single intuitive parameter that has the desirable properties, then we can use it. If not, we can try to find a measure that can be easily computed from an intuitive parameter and that scales linearly with the sequential complexity. The Linpack benchmark, which performs matrix factorization, does this. It is known that when compiled well the benchmark should take $\frac{2n^3}{3}$ operations of a certain type to factorize an n-by-n matrix, and the rest of the operations are either proportional to or completely dominated by these. This number of operations is easily computed from the input matrix dimension n, clearly satisfies the linear scaling property, and used as the measure of work.

Real applications are often more complex, since there are multiple parameters to be scaled. Even so, as long as we have a well-defined rule for scaling them together we may be able to construct a measure of work that is linear in the combined effect of the parameters on sequential complexity. However, such work counts may no longer be simple or intuitive to the user, and they require

either the evaluator to know a lot about the application or the benchmark provider to provide such information. Also, in complex applications analytical predictions are usually simplified (e.g. they are average case, or they do not reflect "implementation" activities that can be quite significant), so the actual growth rates of instructions or operations executed can be a little different than expected.

If none of these mechanisms works, the most general technique is to actually run the best sequential algorithm on a uniprocessor with the base problem and with the scaled problem, and determine how some work measure has grown. If a certain type of high-level operation, such as a particle-particle interaction, is known to always be directly proportional to the sequential complexity, then we can count these operations. More generally, and what we recommend if possible, is to measure the time taken to run the problem on a uniprocessor assuming that all memory references are cache hits and take the same amount of time (say a single cycle), thus eliminating artifacts due to the memory system. This work measure reflects what actually goes on when running the program, yet avoids the thrashing and superlinearity problems, and we call it the *perfect-memory execution time*. Notice that it corresponds very closely to the sequential *busy-useful* time introduced in Section 3.5. Many computers have system utilities that allow us to profile computations to obtain this perfect-memory execution time. If not, we must resort to measuring how many times some high-level operation occurs.

Once we have a work measure, we can compute speedups as described above. However, determining the input configuration that yields the desired execution time may take some iterative refinement.

Memory constrained scaling

This model is motivated by the assumption that the user wants to run the largest problem possible without overflowing the machine's memory, regardless of execution time. For example, it might be important for an astrophysicist to run an N-body simulation with the largest number of particles that the machine can accommodate, to increase the resolution with which the particles sample the universe. An improvement metric that has been used with MC scaling is *scaled speedup*, which is defined as the speedup of the larger (scaled) problem on the larger machine, that is, it is the ratio of the time that the scaled problem would take to run on a single processor to the time that it takes on the scaled machine. Presenting scaled speedups under MC scaling is often attractive to vendors because such speedups tend to be high. This is because what we are really measuring is the problem-constrained speedup on a very large problem, which tends to have a low communication to computation ratio and abundant concurrency. In fact, the problem will very likely not fit in a single node's memory and may take a very long time on a uniprocessor due to thrashing (if it runs at all), so the measured scaled speedup is likely to be highly superlinear. The scaled problem is not what we run on a uniprocessor anyway under this model, so this is not an appropriate speedup metric.

Under MC scaling, neither work nor execution time are held fixed. Using work divided by time as the performance metric as always, we can define speedup as:

$$Speedup_{MC}(p \text{ processors}) = \frac{Work(p \cdot procs)}{Time\langle p \cdot procs\rangle} \times \frac{Time(1 \cdot proc)}{Work(1 \cdot proc)}$$

$$= \frac{IncreaseInWork}{IncreaseInExecutionTime}$$

If the increase in execution time were only due to the increase in work and not due to overheads of parallelism—and if there were no memory system artifacts, which are usually less likely under MC scaling—the speedup would be p, which is what we want. Work is measured as discussed above for TC scaling.

MC scaling is indeed what many users may want to do. Since data set size grows fastest under it compared to other models, parallel overheads grow relatively slowly and speedups often tend to be better than for other scaling models. However for many types of applications MC scaling leads to a serious problem: The execution time—for the parallel execution—can become intolerably large. This problem can occur in any application where the serial execution time grows more rapidly than the memory usage.

Example 4-2   Matrix factorization is a simple example in which the serial execution time grows more rapidly than the memory usage (data set size). Show how MC scaling leads to a rapid increase in parallel execution time for this application.

Answer   While the data-set size and memory usage for an n-by-n matrix grows as $O(n^2)$ in matrix factorization, the execution time on a uniprocessor grows as $O(n^3)$. Assume that a 10000-by-10000 matrix takes about 800 MB of memory and can be factorized in 1 hour on a uniprocessor. Now consider a scaled machine consisting of a thousand processors. On this machine, under MC scaling we can factorize a 320,000-by-320,000 matrix. However, the execution time of the parallel program (even assuming perfect speedup) will now increase to about 32 hours. The user may not be willing to wait this much longer.

Of the three models, time constrained scaling is increasingly recognized as being the most generally viable. However, one cannot claim any model to be the most realistic for all applications and all users. Different users have different goals, work under different constraints, and are in any case unlikely to follow a given model very strictly. Nonetheless, for applications that don't run continuously these three models are useful, comprehensive tools for an analysis of scaled performance. Let us now examine a simple example—the equation solver kernel from Chapter 2—to see how it interacts with different scaling models and how they affect its architecturally relevant behavioral characteristics.

Impact of Scaling Models on the Equation Solver Kernel

For an n-by-n grid, the memory requirement of the simple equation solver is $O(n^2)$. Computational complexity is $O(n^2)$ times the number of iterations to convergence, which we conservatively assume to be $O(n)$ (the number of iterations taken for values to flow from one boundary to the other). This leads to a sequential computational complexity of $O(n^3)$.

Execution Time and Memory Requirements Consider the execution time and memory requirements under the three scaling models, assuming "speedups" due to parallelism equal to p in all cases.

PC Scaling As the same n-by-n grid is divided among more processors p, the memory requirements per processor decrease linearly with p, as does the execution time assuming linear speedup.

TC Scaling By definition, the execution time stays the same. Assuming linear speedup, this means that if the scaled grid size is k-by-k, then $k^3/p = n^3$, so $k = n \times \sqrt[3]{p}$. The amount of memory needed per processor is therefore $k^2/p = n^2/\sqrt[3]{p}$, which diminishes as the cube root of the number of processors.

MC Scaling By definition, the memory requirements per processor stay the same at $O(n^2)$ where the base grid for the single processor execution is n-by-n. This means that the overall size of the grid increases by a factor of p, so the scaled grid is now $n\sqrt{p}$-by-$n\sqrt{p}$ rather than n-by-n. Since it now takes $n\sqrt{p}$ iterations to converge, the sequential time complexity is $O((n\sqrt{p})^3)$. This means that even assuming perfect speedup due to parallelism, the execution time of the scaled problem on p processors is $O\left(\frac{(n\sqrt{p})^3}{p}\right)$ or $n^3\sqrt{p}$. Thus, the parallel execution time is greater than the sequential execution time of the base problem by a factor of $\sqrt{p}$, growing with the square root of the number of processors. Even under the linear speedup assumption, a problem that took 1 hour on one processor takes 32 hours on a 1024-processor machine under MC scaling.

For this simple equation solver, then, the execution time increases quickly under MC scaling, and the memory requirements per processor decrease under TC scaling.

Execution Characteristics Let us consider the effects of different scaling models on the concurrency, communication-to-computation ratio, synchronization and I/O frequency, and temporal and spatial locality. Let us examine each.

Concurrency. The concurrency in this kernel is proportional to the number of grid points. It remains fixed at n^2 under PC scaling, grows proportionally to p under MC scaling, and grows proportionally to $p^{0.67}$ under TC scaling.

Communication to computation ratio The communication to computation ratio is the perimeter to area ratio of the grid partition assigned to each processor; that is, it is inversely proportional to the square root of the number of points per processor. Under PC scaling, the ratio grows as $\sqrt{p}$. Under MC scaling the size of a partition does not change, so neither does the communication to computation ratio. Finally, under TC scaling, the size of a processor's partition diminishes as the cube root of the number of processors, so the ratio increases as the sixth root of p.

Synchronization and I/O Frequency The equation solver synchronizes at the end of every grid sweep, to determine convergence. Suppose that it also performed I/O then, e.g. outputting the maximum error at the end of each sweep. Under PC scaling, the work done by each processor in a given sweep decreases linearly as the number of processors increases, so assuming linear speedup the frequency of synchronization and I/O grows linearly with p. Under MC scaling the frequency remains fixed, and under TC scaling it increases as the cube root of p.

Working Set Size (Temporal Locality) The size of the important working set in this solver is exactly the size of a processor's partition of the grid. Thus, it and the cache requirements diminish linearly with p under PC scaling, stay constant under MC scaling, and diminish as the cube root of p under TC scaling. Thus, although the problem size grows under TC scaling, the working set size of each processor diminishes.

Spatial Locality Spatial locality is best within a processor's partition and at row-oriented boundaries, and worst at column-oriented boundaries. Thus, it decreases as a processor's partition becomes smaller and column-oriented boundaries become larger relative to partition area. It therefore remains just as good under MC scaling, decreases quickly under PC scaling, and decreases less quickly under TC scaling.

Message Size in Message Passing An individual message is likely to be a border row or column of a processor's partition which is the square root of partition size. Hence message size here scales similarly to the communication to computation ratio.

It is clear from the above that we should expect the lowest parallelism overhead and highest speedup as defined above under MC scaling, next under TC scaling, and that we should expect speedups to degrade quite quickly under PC scaling, at least once the overheads start to become significant relative to useful work. It is also clear that the choice of application parameters as well as the scaling model affect architectural interactions with the extended memory hierarchy such as spatial and temporal locality. Unless it is known that a particular scaling model is the right one for an application, or that one is inappropriate, for all the reasons discussed in this section it is appropriate to evaluate a machine under all three scaling models. We shall see the interactions with architectural parameters and their importance for evaluation in more detail in the next couple of sections, where we discuss actual evaluations. First, let us take a brief look at the other important but more subtle aspect of scaling: how to scale application parameters to meet the constraints of a given scaling model.

4.2.4 Scaling Workload Parameters

Let us now take away our simplifying assumption that a workload has only a single parameter. For instance, the Ocean application has a vector of four parameters: *n, ε, Δt, T,* that the simple equation solver kernel does not expose. Scaling workload parameters is not an issue under PC scaling. However, under TC or MC scaling, how should the parameters of a workload be scaled to meet the prescribed time or memory constraints? The main point to be made here is that the different parameters are often related to one another, and it may not make sense to scale only one of them without scaling others, or to scale them independently. Each parameter may make its own unique contribution to a given execution characteristic (time, memory usage, communication to computation ratio, working sets, etc.). For example, in the Barnes-Hut application the execution time grows not just as $n\log n$ but as $\frac{1}{\theta^2 \Delta t} n\log n$, where θ is the force-calculation accuracy parameter and Δt is the physical interval between time-steps; both θ and Δt should be scaled as n changes. Even the simple equation solver kernel has another parameter ε, which is the tolerance used to determine convergence of the solver. Making this tolerance smaller—as should be done as n scales in a real application—increases the number of iterations needed for convergence, and hence increases execution time without affecting memory requirements. Thus the grid size, memory requirements and working set size would increase much more slowly under TC scaling, the communication to computation ratio would increase more quickly, and the execution time would increase more quickly under MC scaling. As architects using workloads, it is very important that we understand the relationships among parameters *from an application user's viewpoint*, and scale the parameters in our evaluations according to this understanding. Otherwise we are liable to arrive at incorrect architectural conclusions.

The actual relationships among parameters and the rules for scaling them depend on the domain and the application. There are no universal rules, which makes good evaluation even more interesting. For example, in scientific applications like Barnes-Hut and Ocean, the different application parameters usually govern different sources of error in the accuracy with which a physical phenomenon (such as galaxy evolution) is simulated; appropriate rules for scaling these parameters together are therefore driven by guidelines for scaling different types of error. Ideally, benchmark suites will describe the scaling rules—and may even encode them in the application, leaving only one free parameter—so the architect does not have to worry about learning them. Exercises 4.11 and 4.12 illustrate the importance of proper application scaling. We will see that scaling parameters appropriately can often lead to quantitatively and sometimes even qualitatively different architectural results than scaling only the data set size parameter n.

4.3 Evaluating a Real Machine

Having understood the importance of scaling problem size with machine size, and the effects that problem and machine size have on fundamental behavioral characteristics and architectural interactions, we are now ready to develop specific guidelines for the two major types of workload-driven evaluation: evaluating a real machine, and evaluating an architectural idea or tradeoff. Evaluating a real machine is in many ways simpler: the machine characteristics and parameters are fixed, and all we have to worry about is choosing appropriate workloads and workload parameters. When evaluating an architectural idea or tradeoff in a general context, organizational and performance parameters of the architecture can become variable as well, and the evaluation is usually done through simulation, so the set of issues to consider becomes larger. This section provides a prescriptive template for evaluating a real machine. The use of microbenchmarks to isolate performance characteristics is discussed first. Then, we discuss the major issues in choosing workloads for an evaluation. This is followed by guidelines for evaluating a machine once a workload is chosen, first when the number of processors is fixed, and then when it is allowed to be varied. The section concludes with a discussion of appropriate metrics for measuring the performance of a machine and for presenting the results of an evaluation. All these issues in evaluating a real machine, discussed in this section, are relevant to evaluating an architectural idea or tradeoff as well; the next section builds upon this discussion, covering the additional issues that arise in the latter type of evaluation.

4.3.1 Performance Isolation using Microbenchmarks

As a first step in evaluating a real machine, we might like to understand its basic performance capabilities; that is, the performance characteristics of the primitive operations provided by the programming model, communication abstraction, or hardware-software interface. This is usually done with small, specially written programs called *microbenchmarks* [SGC93], that are designed to isolate these performance characteristics (latencies, bandwidths, overheads etc.).

Five types of microbenchmarks are used in parallel systems, the first three of which are used for uniprocessor evaluation as well:

Processing microbenchmarks measure the performance of the processor on non memory-referencing operations such as arithmetic operations, logical operations and branches.

Local memory microbenchmarks detect the organization, latencies and bandwidths of the levels of the cache hierarchy and main memory within the node, and measure the performance of

local read and write operations satisfied at different levels, including those that cause TLB misses and page faults.

Input-output microbenchmarks measure the characteristics of I/O operations such as disk reads and writes of various strides and lengths.

Communication microbenchmarks measure data communication operations such as message sends and receives or remote loads and stores of different types, and

Synchronization microbenchmarks measure the performance of different types of synchronization operations such as locks and barriers.

The communication and synchronization microbenchmarks depend on the communication abstraction or programming model used. They may involve one or a pair of processors— e.g. a single remote read miss, a send-receive pair, or the acquisition of a free lock—or they may be collective, such as broadcast, reduction, all-to-all communication, probabilistic communication patterns, many processors contending for a lock, or barriers. Different microbenchmarks may be designed to stress uncontended latency, bandwidth, overhead and contention.

For measurement purposes, microbenchmarks are usually implemented as repeated sets of the
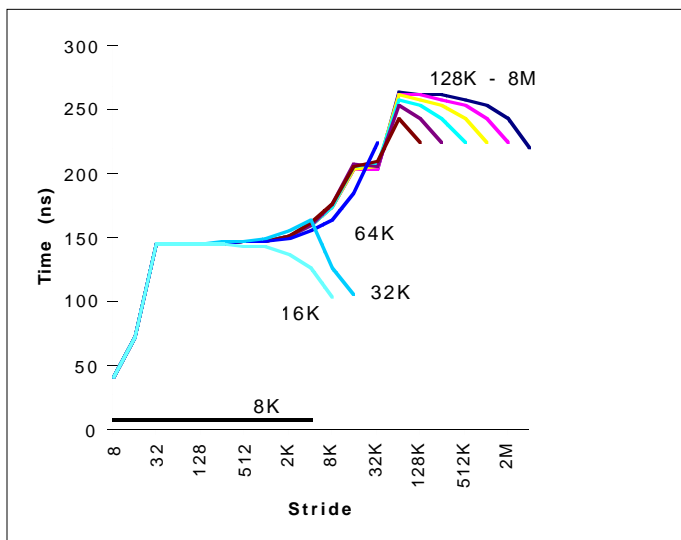
The Microbenchmark

for times ← 0 **to** 10,000 **do**
    for i ← 0 **to** ArraySize-1 **by** stride **do**
        load A[i];

Figure 4-3 Results of a microbenchmark experiment on a single processing node of the Cray T3D multiprocessor.

The microbenchmark consists of a large number of loads from a local array. The Y-axis shows the time per load in nanoseconds. The X-axis is the stride between successive loads in the loop, i.e. the difference in the addresses of the memory locations being accessed. And the different curves correspond to the size of the array (ArraySize) being strided through. When ArraySize is less than 8KB, the array fits in the processor cache so all loads are hits and take 6.67 ns to complete. For larger arrays, we see the effects of cache misses. The average access time is the weighted sum of hit and miss time, until there is an inflection when the stride is longer than a cache block (32 words or 128 bytes) and every reference misses. The next rise occurs due to some references causing page faults, with an inflection when the stride is large enough (16KB) for every consecutive reference does so. The final rise is due to conflicts at the memory banks in the 4-bank main memory, with an inflection at 64K stride when consecutive references hit the same bank.

primitive operations, e.g. ten thousand remote reads in a row. They often have simple parameters that can be varied to obtain a fuller characterization. For example, the number of processors used may be a parameter to a microbenchmark that measures the performance of a collective communication operation among a group of processors. Similarly, the stride between consecutive reads may be a parameter to a local memory microbenchmark, and may be set to ensure that subse-

quent reads map to the same cache block or that they fall on different pages. Figure 4-3 shows a typical profile of a machine obtained using a local memory microbenchmark. As mentioned above, the role of microbenchmarks is to isolate and understand the performance of basic system capabilities. A more ambitious hope, not achieved so far, is that if workloads can be characterized as weighted sums of different primitive operations, then a machine's performance on a given workload can be predicted from its performance on the corresponding microbenchmarks. We will discuss more specific microbenchmarks when we measure real systems in later chapters.

Having isolated the performance characteristics, the next step is to evaluate the machine on more realistic workloads. There are three major axes we must navigate: the workloads, their problem sizes, and the number of processors (or the machine size). Lower level machine parameters, such as organizational granularities and the performance parameters of the communication architecture, are fixed when evaluating a real machine. We begin by discussing the issues in choosing workloads. Then, assuming that we have chosen a workload, we will examine how to evaluate a machine with it. For simplicity, we will first assume that another axis, the number of processors, is fixed as well, and use our understanding of parameter interactions to see how to choose problem sizes to evaluate the fixed-size machine. Finally, we will discuss varying the number of processors as well, using the scaling models we have just developed.

4.3.2 Choosing Workloads

Beyond microbenchmarks, workloads or benchmarks used for evaluation can be divided into three classes in increasing order of realism and complexity: kernels, applications, and multiprogrammed workloads. Each has its own role, advantages and disadvantages.

Kernels These are well-defined parts of real applications, but are not complete applications themselves. They can range from simple kernels such as a parallel matrix transposition or a simple near-neighbor grid sweep, to more complex, substantial kernels that dominate the execution times of their applications. Examples of the latter from scientific computing include matrix factorization and iterative methods to solve partial differential equations, such as our equation solver kernel, while examples from information processing may include complex database queries used in decision support applications or sorting a set of numbers in ascending order. Kernels expose higher-level interactions that are not present in microbenchmarks, and as a result lose a degree of performance isolation. Their key property is that their performance-relevant characteristics—communication to computation ratio, concurrency, and working sets, for example—can be easily understood and often analytically determined, so that observed performance as a result of the interactions can be easily explained in light of these characteristics.

Complete Applications Complete applications consist of multiple kernels, and exhibit higher-level interactions among kernels that an individual kernel cannot reveal. The same large data structures may be accessed in different ways by multiple kernels in an application, and different data structures accessed by different kernels may interfere with one another in the memory system. In addition, the data structures that are optimal for a kernel in isolation may not be the best ones to use in the complete application, which must strike a balance among the needs of its different parts. The same holds for partitioning techniques. If there are two independent kernels, then we may decide to not partition each among all processes, but rather to share processes among them. Different kernels that share a data structure may be partitioned in ways that strike a between their different access patterns to the data, leading to the maximum overall locality. The presence of multiple kernels in an application introduces many subtle interactions, and the per-

formance-related characteristics of complete applications usually cannot be exactly determined analytically.

Multiprogrammed Workloads These consist of multiple sequential and parallel applications running together on the machine. The different applications may either time-share the machine or *space-share* it (different applications running on disjoint subsets of the machine's processors) or both, depending on the operating system's multiprogramming policies. Just as whole applications are complicated by higher level interactions among the kernels that comprise them, multiprogrammed workloads involve complex interactions among whole applications themselves.

As we move from kernels to complete applications and multiprogrammed workloads, we gain in realism, which is very important. However, we lose in our ability to describe the workloads concisely, to explain and interpret the results unambiguously, and to isolate performance factors. In the extreme, multiprogrammed workloads are difficult to design: Which applications should be included in such a workload and in what proportion?. It is also difficult to obtain repeatable results from multiprogrammed workloads due to subtle timing-dependent interactions with the operating system. Each type of workload has its place. However, the higher-level interactions exposed only by complete applications and multiprogrammed workloads, and the fact that they are the workloads that will actually be run on the machine by users, make it important that we use them to ultimately determine the overall performance of a machine, the value of an architectural idea, or the result of a tradeoff. Let us examine some important issues in choosing such workloads (applications, multiprogrammed loads and even complex kernels) for an evaluation.

The desirable properties of a set of workloads are representativeness, coverage of behavioral properties, and adequate concurrency.

Representativeness of Application Domains

If we are performing an evaluation as users looking to procure a machine, and we know that the machine will be used to run only certain types of applications (say databases, transaction processing, or ocean simulation), then this part of our job is easy. On the other hand, if our machine may be used to run a wide range of workloads, or if we are designers trying to evaluate a machine to learn lessons for the next generation, we should choose a mix of workloads representative of a wide range of domains. Some important domains for parallel computing today include *scientific* applications that model physical phenomena; *engineering* applications such as those in computer-aided design, digital signal processing, automobile crash simulation, and even simulations used to evaluate architectural tradeoffs; *graphics* and visualization applications that render scenes or volumes into images; *media processing* applications such as image, video and audio analysis and processing, speech and handwriting recognition; *information management* applications such as databases and transaction processing; *optimization* applications such as crew scheduling for an airline and transport control; *artificial intelligence* applications such as expert systems and robotics; *multiprogrammed* workloads; and the *operating system* itself, which is a particularly complex parallel application.

Coverage of Behavioral Properties

Workloads may vary substantially in terms of the entire range of performance related characteristics discussed in Chapter 3. As a result, a major problem in evaluation is that it is very easy to lie with or be misled by workloads. For example, a study may choose workloads that stress the fea-

ture for which an architecture has an advantage (say communication latency), but do not exercising aspects it performs poorly (say contention or communication bandwidth). For general-purpose evaluation, it is important that the workloads we choose taken together stress a range of important performance characteristics. For example, we should choose workloads with low and high communication to computation ratios, small and large working sets, regular and irregular access patterns, and localized and collective communication. If we are interested in particular architectural characteristics, such as aggregate bandwidth for all-to-all communication among processors, then we should choose at least some workloads that stress those characteristics.

Also, real parallel programs will not always be highly optimized for good performance along the lines discussed in Chapter 3, not just for the specific machine at hand but even in more generic ways. This may be either because the effort involved in optimizing programs is more than the user is willing to expend, or because the programs are generated with the help of automated tools. The level of optimization can greatly affect important execution characteristics. In particular, there are three important levels to consider:

Algorithmic The decomposition and assignment of tasks may be less than optimal, and certain algorithmic enhancements for data locality such as blocking may not be implemented; for example, strip-oriented versus block-oriented assignment for a grid computation (see Section 2.4.3).

Data structuring The data structures used may not interact optimally with the architecture, causing artifactual communication; for example, two-dimensional versus four-dimensional arrays to represent a two-dimensional grid in a shared address space (see Section 3.4.1).

Data layout, distribution and alignment Even if appropriate data structures are used, they may not be distributed or aligned appropriately to pages or cache blocks, causing artifactual communication in shared address space systems.

Where appropriate, we should therefore evaluate the robustness of machines or features to workloads with different levels of optimization.

Concurrency

The dominant performance bottleneck in a workload may be the computational load imbalance, either inherent to the partitioning method or due to the way synchronization is orchestrated (e.g. using barriers instead of point to point synchronization). If this is true, then the workload may not be appropriate for evaluating a machine's communication architecture, since there is little that the architecture can do about this bottleneck: Even great improvements in communication performance may not affect overall performance much. While it is useful to know what kinds of workloads and problem sizes do not afford enough concurrency for the number of processors at hand—since this may limit the applicability of a machine of that size—to evaluate communication architectures we should ensure that our workloads have adequate concurrency and load balance. A useful concept here is that of *algorithmic speedup*. This is the speedup—according to the scaling model used—assuming that all memory references and communication operations take zero time. By completely ignoring the performance impact of data access and communication, algorithmic speedup measures the computational load balance in the workload, together with the extra work done in the parallel program. This highly idealized performance model is called a PRAM or parallel random access machine [FoW78] (albeit completely unrealistic, it has in fact traditionally been most widely used to evaluate the complexity of parallel algorithms). In general, we should isolate performance limitations due to workload characteristics that a machine cannot

do much about from those that it can. It is also important that the workload take long enough to run to be interesting on a machine of the size being evaluated, and be realistic in this sense, though this is often more a function of the input problem size than inherently of the workload itself.

Many efforts have been made to define standard benchmark suites of parallel applications to facilitate workload-driven architectural evaluation. They cover different application domains and have different philosophies, and some of them are described in APPENDIX A. While the workloads that will be used for the illustrative evaluations in the book are a very limited set, we will keep the above criteria in mind in choosing them. For now, let us assume that a parallel program has been chosen as a workload, and see how we might use it to evaluate a real machine. We first keep the number of processors fixed, which both simplifies the discussion and also exposes the important interactions more cleanly. Then, we discuss varying the number of processors as well.

4.3.3 Evaluating a Fixed-Size Machine

Since the other major axes in evaluating a real machine (the programs in the workload and the machine size) are fixed, the only variable here is the parameters the workload. In discussing the implications of scaling models, we saw how these application parameters interact with the number of processors to affect some of the most important execution characteristics of the program. While the number of processors is fixed here, changing problem size itself can dramatically affect all these characteristics and hence the results of an evaluation. In fact, it may even change the nature of the dominant bottleneck; i.e. whether it is communication, load imbalance or local data access. This already tells us that *it is insufficient to use only a single problem size in an evaluation*, even when the number of processors is fixed.

We can use our understanding of parameter implications to actually choose problem sizes for a study, if we extend it to include not only behavioral characteristics of the program but also how they interact with organizational parameters of the extended memory hierarchy. Our goal is to obtain adequate coverage of realistic behaviors while at the same time restricting the number of problem sizes we need. We do this in a set of structured steps, in the process demonstrating the pitfalls of choosing only a single size or a narrow range of sizes. The simple equation solver kernel will be used to illustrate the steps quantitatively, even though it is not a full-fledged workload. For the quantitative illustration, let us assume that we are evaluating a cache-coherent shared address space machine with 64 single-processor nodes, each with 1 MB of cache and 64MB of main memory. The steps are as follows.

1. Appeal to higher powers. The high-level goals of the study may choose the problem sizes for us. For example, we may know that users of the machine are interested in only a few specified problem sizes. This simplifies our job, but is uncommon and not a general-purpose methodology. It does not apply to the equation solver kernel.

2. Determine a range of problem sizes. Knowledge of real usage may identify a range below which problems are unrealistically small for the machine at hand and above which the execution time is too large or users would not want to solve problems. This too is not particularly useful for the equation solver kernel. Having identified a range, we can go on to the next step.

3. Use inherent behavioral characteristics. Inherent behavioral characteristics help us choose problem sizes within the selected range such as communication to computation ratio and load

balance. Since the inherent communication to computation ratio usually decreases with increasing data set size, large problems may not stress the communication architecture enough—at least with inherent communication—while small problems may over-stress it unrepresentatively and potentially hide other bottlenecks. Similarly, concurrency often increases with data set size, giving rise to a tradeoff we must navigate: We would like to choose at least some sizes that are large enough to be load balanced but not so large that the inherent communication becomes too small. The size of the problem relative to the number of processors may also affect the fractions of execution time spent in different phases of the application, which may have very different load balance, synchronization and communication characteristics. For example, in the Barnes-Hut application case study smaller problems cause more of the time to be spent in the tree-building phase, which doesn't parallelize very well and has much worse properties than the force-calculation phase which usually dominates in practice. We should be careful not to choose unrepresentative scenarios in this regard.

Example 4-3   How would you use the above inherent behavioral characteristics to choose problem sizes for the equation solver kernel?

Answer   For this kernel, enough work and load balance might dictate that we have partitions that are at least 32-by-32 points. For a machine with 64 (8x8) processors, this means a total grid size of at least 256-by-256. This grid size requires the communication of 4*32 or 128 grid elements in each iteration, for a computation of 32*32 or 1K points. At five floating point operations per point and 8 bytes per grid point, this is an inherent communication to computation ratio of one byte every five floating point operations. Assuming a processor that can deliver 200 MFLOPS on this calculation, this implies a bandwidth of 40MB/s. This is quite small for modern multiprocessor networks, even despite its burstiness. Let us assume that below 5MB/s communication is asymptotically small for our system. Then, from the viewpoint of inherent communication to computation ratio there is no need to run problems larger than 256-by-256 points (only 64K*8B or 512KB) per processor, or 2K by 2K grids overall.

Finally, we proceed to the most complex step:

4. Use temporal and spatial locality interactions. These inherent characteristics vary smoothly with problem size, so to deal with them alone we can pick a few sizes that span the interesting spectrum. If their rate of change is very slow, we may not need to choose many sizes. Experience shows that about three is a good number in most cases. For example, for the equation solver kernel we might have chosen 256-by-256, 1K-by-1K, and 2K-by-2K grids. Unlike the previous properties, the interactions of temporal and spatial locality with the architecture exhibit thresholds in their performance effects, including the generation of artifactual communication as problem size changes. We may need to extend our choice of problem sizes to obtain enough coverage with respect to these thresholds. At the same time, the threshold nature can help us prune the parameter space. Let us look separately at effects related to temporal and spatial locality.

Temporal locality and working sets

Working sets fitting or not fitting in a local replication store can dramatically affect execution characteristics such as local memory traffic and artifactual communication, even if the inherent

communication and computational load balance do not change much. In applications like Ray-trace, the important working sets are large and consist of data that are mostly assigned to remote nodes, so artifactual communication due to limited replication capacity may dominate inherent communication. Unlike inherent communication this tends to grow rather than diminish with increasing problem size. We should include problem sizes that represent both sides of the threshold (fitting and not fitting) for the important working sets, if these problem sizes are realistic in practice. In fact, when realistic for the application we should also include a problem size that is very large for the machine, e.g. such that it almost fills the memory. Large problems often exercise architectural and operating system interactions that smaller problems do not, such as TLB misses, page faults, and a large amount of traffic due to cache capacity misses. The following idealized example helps illustrate how we might choose problem sizes based on working sets.

Example 4-4   Suppose that an application has the idealized miss rate versus cache size curve shown in Figure 4-4(a) for a fixed problem size and number of processors, and for the cache organization of our machine. If C is the machine's cache size, how should
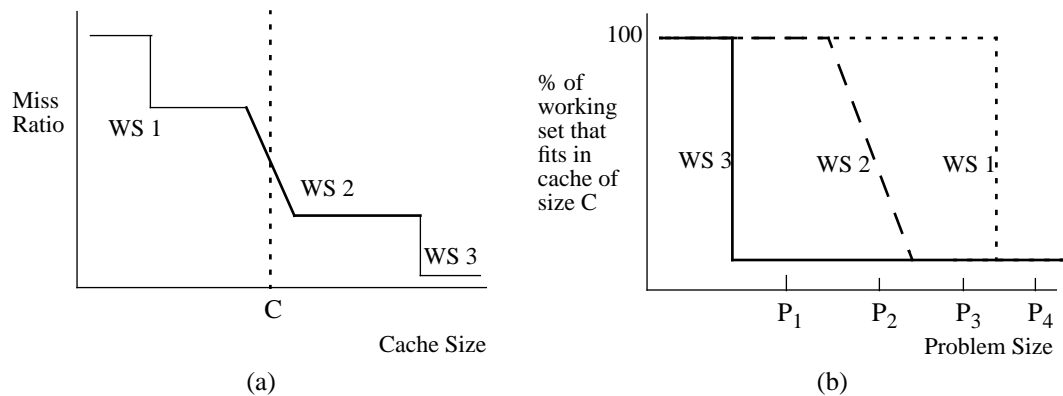
Figure 4-4 Choosing problem sizes based on working sets fitting in the cache.

The graph on the left shows the working set curve (miss rate versus cache size) for a fixed problem size with our chosen number of processors. C is the cache size of our machine. This curve identifies three knees or working sets, two very sharply defined and one less so. The graph on the right shows, for each of the working sets, a curve depicting whether or not they fit in the cache of size C as the problem size increases. A knee in a curve represents the problem size at which that working set no longer fits. We can see that a problem of size P_1 fits WS1 and WS2 but not WS3, problem P_2 fits WS1 and part of WS2 but not WS3, P_3 fits WS1 only, while P_4 does not fit any working set in the cache.

this curve influence the choice of problem sizes used to evaluate the machine?

Answer   We can see that for the problem size (and number of processors) shown, the first working set fits in the cache of size C, the second fits only partially, and the third does not fit. Each of these working sets scales with problem size in its own way. This scaling determines at what problem size that working set might no longer fit in a cache of size C, and therefore what problem sizes we should choose to cover the

representative cases. In fact, if the curve truly consists of sharp knees, then we can draw a different type of curve, this time one for each important working set. This curve, shown in Figure 4-4(b), depicts whether or not that working set fits in our cache of size *C* as the problem size changes. If the problem size at which a knee in this curve occurs for an important working set is within the range of problem sizes that we have already determined to be realistic, then we should ensure that we include a problem size on each side of that knee. Otherwise, we should ignore the unrealistic side. Not doing this may cause us to either miss important effects related to stressing the memory or communication architecture, or observe substantial effects that are not representative of reality. The fact that the curves are flat on both sides of a knee in this example means that if all we care about from the cache is the miss rate then we need to choose only one problem size on each side of each knee for this purpose, and prune out the rest.[1]

We shall discuss the use of working sets in more detail when we examine choosing parameters for a simulation study in the next section, where the cache size becomes a variable as well.

Example 4-5   How might working sets influence our choice of problem sizes for the equation solver?

Answer   The most important working sets for the equation solver are encountered when a couple of subrows of a partition fit in the cache, and when a processor's entire partition fit in the cache. Both are very sharply defined in this simple kernel. Even with the largest grid we chose based on inherent communication to computation ratio (2K-by-2K), the data set size per processor is only 0.5MB, so both these working sets fit comfortably in the cache (if a 4-d array representation is used, there are essentially no conflict misses). Two subrows not fitting in a 1MB cache would imply a subrow of 64K points, so a total grid of 64*8 or 512K-by-512K points. This is a data set of 32GB per processor, which is far too large to be realistic. However, having the other important working set—the whole partition—not fit in a 1MB cache is realistic. It leads to either a lot of local memory traffic or a lot of artifactual communication, if data are not placed properly, and we would like to represent such a situation. We can do this by choosing one problem size such that its working set (here partition size) is larger than 1MB per processor, e.g. 512 -by-512 points (2MB) per processor or 4K-by-4K points overall. This does not come close to filling the machine's memory, so we should choose one more problem size for that

---

1. Pruning a flat region of the miss rate curve is not necessarily appropriate if we also care about other aspects of cache behavior than miss rate which are also affected by cache size. We shall see an example when we discuss tradeoffs among cache coherence protocols in Chapter 5.

purpose, say 16K-by-16K points overall or 32MB per processor. We now have five problem sizes: 256-by-256, 1K-by-1K, 2K-by-2K, 4K-by-4K and 16K-by-16K.

Spatial locality

Consider the equation solver again, and suppose that the data structure used to represent the grid in a shared address space is a two-dimensional array. A processor's partition, which is its important working set, may not remain in its cache across grid sweeps, due to either limited capacity or cache conflicts which may be quite frequent now that the subrows of a partition are not contiguous in the address space (the non-contiguity and its effects will be examined again in Chapters 5 and 8). If so, it is important that a processor's partition be allocated in its local memory on a distributed-memory machine. The granularity of allocation in main memory is a page, which is typically 4-16KB. If the size of a subrow is less than the page size, proper allocation becomes very difficult and there will be a lot of artifactual communication. However, as soon as that threshold in size is crossed, allocation is not a problem anymore and there is little artifactual communication. Both situations may be realistic, and we should try to represent both. If the page size is 4KB, then our first three problems have a subrow smaller than 4KB so they cannot be distributed properly, while the last two have subrows greater than or equal to 4K so can be distributed well provided the grid is aligned to a page boundary. We do not need to expand our set of problem sizes for this purpose.

A more stark example of spatial locality interactions is found in a program and an architectural interaction that we have not yet discussed. The program, called Radix, is a sorting program that will be described later in this chapter, and the architectural interaction, called false sharing, will be discussed in Chapter 5. However, we show the result to illustrate the importance of considering spatial interactions in our choice of problem sizes. Figure 4-5 shows how the miss rate for this program running on a cache-coherent shared address space machine changes with cache block size, for two different problem sizes n (sorting 256K integers and 1M integers) using the same number of processors p. The false sharing component of the miss rate leads to a lot of artifactual communication, and in this application can completely destroy performance. Clearly, for the given cache block size on our machine, false sharing may or not destroy the performance of radix sorting depending simply on the problem size we choose. It turns out that for a given cache block size false sharing is terrible if the ratio of problem size to number of processors is smaller than a certain threshold, and almost non-existent if it is bigger. Some applications display these threshold effects in spatial locality interactions with problem size, others do not. Identifying the presence of such thresholds requires a deep enough understanding of the application's locality and its interaction with architectural parameters, and illustrates some of the subtleties in evaluation.

The simple equation solver summarizes the dependence of most execution characteristics on problem size, some exhibiting knees on threshold in interaction with architectural parameters and some not. With n-by-n grids and p processes, if the ratio n/p is large, communication to computation ratio is low, the important working sets are unlikely to fit in the processor caches leading to a high capacity miss rate, and spatial locality is good even with a two-dimensional array representation. The situation is completely the opposite when n/p is small: high communication to computation ratio, poor spatial locality and false sharing, and few local capacity misses. The dominant performance bottleneck thus changes from local access in one case to communication in the other. Figure 4-6 illustrates these effects for the Ocean application as a whole, which uses kernels similar to the equation solver.
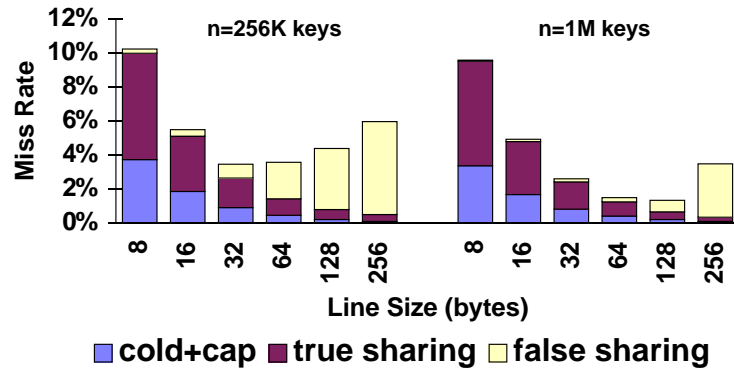
Figure 4-5 Impact of problem size and number of processors on the spatial locality behavior of Radix sorting.

The miss rate is broken down into cold/capacity misses, true sharing (inherent communication) misses, and misses due to false shar-
ing of data. As the block size increases for a given problem size and number of processors, there comes a point when the ratio men-
tioned in the text becomes smaller than the block size, and substantial false sharing is experienced. This is clearly a threshold effect,
and happens at different block sizes for different problem sizes. A similar effect would have been observed if the problem size were
kept constant and the number of processors changed.
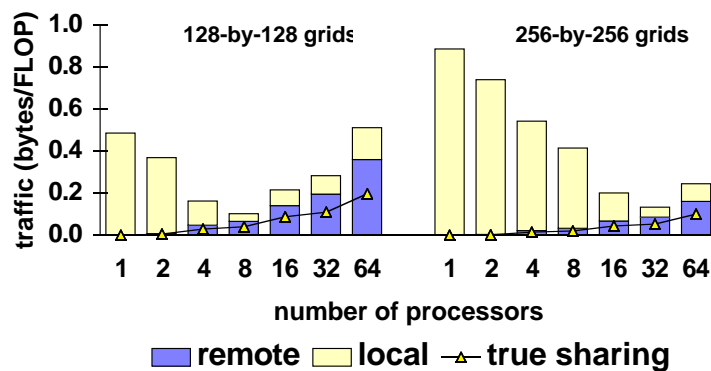
Figure 4-6 Effects of problem size, number of processors and working set fitting in the cache.

The figure shows the effects on the memory behavior of the Ocean application in a shared address space. The traffic (in bytes per
floating point operation or FLOP) is broken down into cold/capacity traffic, true sharing (inherent communication) traffic, and traffic
due to false sharing of data. The true sharing (inherent communication) traffic increases with the number of processors, and decreases
from the smaller problem to the larger. As the number of processors increases for a given problem size, the working set starts to fit in
the cache and a domination by local misses is replaced by a domination by communication. This change occurs at a larger number of
processors for the larger problem, since the working set is proportional to n^2/p. If we focus on the 8-processor breakdown for the two
problem sizes, we see that for the small problem the traffic is dominantly remote (since the working set fits in the cache) while for the
larger problem it is dominantly local (since the working set does not fit in the cache),

While there are no absolute formulae for choosing problem sizes to evaluate a machine, and the
equation solver kernel is a trivial example, the above steps are useful guidelines. We should also

ensure that the results we obtain for a machine are not due to artifacts that can be easily removed in the program, including systematic cache conflict misses and some kinds of contention. These may be useful for evaluating the robustness of the machine, but should not be the only situations we evaluate. Despite the number of issues to consider, experience shows that the number of problems sizes needed to evaluate a fixed-size machine with an application is usually quite small. If we are to compare two machines, it is useful to choose problem sizes that exercise the above scenarios on both machines.

4.3.4 Varying Machine Size

Now suppose we want to evaluate the machine as the number of processors changes. We have already seen how we might scale the problem size, under different scaling models, and what metrics we might use for performance improvement due to parallelism. The issue that remains is how to choose a fixed problem size, at some machine size, as a starting point from which to scale. One strategy is to start from the problem sizes we chose above for a fixed number of processors and then scale them up or down according to the different scaling models. We may narrow down our range of base problem sizes to three: a small, a medium and a large, which with three scaling models will result in nine sets of performance data and speedup curves. However, it may require care to ensure that the problem sizes, when scaled down, will stress the capabilities of smaller machines.

A simpler strategy is to start with a few well-chosen problem sizes on a uniprocessor and scale up from there under all three models. Here too, it is reasonable to choose three uniprocessor problem sizes. The small problem should be such that its working set fits in the cache on a uniprocessor, and its communication to computation ratio exercises the communication architecture even for a relatively small number of processors. This problem will not be very useful under PC scaling on large machines, but should remain fine under TC and at least MC scaling. The large problem should be such that its important working set does not fit in the cache on a uniprocessor, if this is realistic for the application. Under PC scaling the working set may fit at some point (if it shrinks with increasing number of processors), while under MC scaling it is likely not to and to keep generating capacity traffic. A reasonable choice for a large problem is one that fills most of the memory on a single node, or takes a large amount of time on it. Thus, it will continue to almost fill the memory even on large systems under MC scaling. The medium sized problem can be chosen to be in between in some judicious way; if possible, even it should take a substantial amount of time on the uniprocessor. The outstanding issue is how to explore PC scaling for problem sizes that don't fit in a single node's memory. Here the solution is to simply choose such a problem size, and measure speedup relative to a number of processors for which the problem fits in memory.

4.3.5 Choosing Performance Metrics

In addition to choosing parameters, an important question in evaluating or comparing machines is the metrics that should be chosen for the evaluation. Just as it is easy in parallel computing to mislead by not choosing workloads and parameters appropriately, it is also easy to convey the wrong impression by not measuring and presenting results in meaningful ways. In general, both cost and performance are important metrics. In comparing two machines, it is not just their performance that matters but also their cost. And in evaluating how well a machine scales as resources (for example processors and memory) are added, it is not only how performance increases that matters but also how cost increases. Even if speedup increases much less than lin-

early, if the cost of the resources needed to run the program don't increase much more quickly than that then it may indeed be cost-effective to use the larger machine [WH95]. Overall, some measure of "cost-performance" is more appropriate than simply performance. However, cost and performance can be measured separately, and cost is very dependent on the marketplace. The focus here is on metrics for measuring performance.

We have already discussed absolute performance and performance improvement due to parallelism and argued that they are both useful metrics. Here, we first examine some subtler issues in using these metrics to evaluate and especially compare machines, and then understand the role of other metrics that are based on *processing rate* (e.g. megaflops), *resource utilization*, and *problem size*. As we shall see, some metrics are clearly important and should always be presented, while the utility of others depends on what we are after and the environment in which we operate. After discussing metrics, we will discuss some general issues regarding the presentation of results in Section 4.3.6. We focus on issues that pertain especially to multiprocessors.

Absolute Performance

As discussed earlier, to a user of a system the absolute performance is the performance metric that matters the most. Suppose that execution time is our absolute performance metric. Time can be measured in different ways. First, there is a choice between *user time* and *wall-clock time* for a workload. User time is the time the machine spent executing code from the particular workload or program in question, thus excluding system activity and other programs that might be time-sharing the machine, while wall-clock time is the total elapsed time for the workload—including all intervening activity—as measured by a clock hanging on the wall. Second, there is the issue of whether to use the average or the maximum execution time over all processes of the program.

Since users ultimately care about wall-clock time, we must measure and present this when comparing systems. However, if other user programs—not just the operating system—interfere with a program's execution due to multiprogramming, then wall-clock time does not help understand performance bottlenecks in the particular program of interest. Note that user time for that program may also not be very useful in this case, since interleaved execution with unrelated processes disrupts the memory system interactions of the program as well as its synchronization and load balance behavior. We should therefore always present wall-clock time and describe the execution environment (batch or multiprogrammed), whether or not we present more detailed information geared toward enhancing understanding.

Similarly, since a parallel program is not finished until the last process has terminated, it is the time to this point that is important, not the average over processes. Of course, if we truly want to understand performance bottlenecks we would like to see the execution profiles of all processes—or a sample—broken down into different components of time (refer back to Figure 3-12 on page 162, for example). The components of execution time tell us why one system outperforms another, and also whether the workload is appropriate for the investigation under consideration (e.g. is not limited by load imbalance).

Performance Improvement or Speedup

We have already discussed the utility of both absolute performance and performance improvement or speedup in obtaining insights into the performance of a parallel machine. The question

that was left open in measuring speedup for any scaling model is what the denominator in the speedup ratio—performance on one processor—should actually measure. There are four choices:

(1) Performance of the parallel program on one processor of the parallel machine

(2) Performance of a sequential implementation of the same algorithm on one processor of the parallel machine

(3) Performance of the "best" sequential algorithm and program for the same problem on one processor of the parallel machine

(4) Performance of the "best" sequential program on an agreed-upon standard machine.

The difference between (1) and (2) is that the parallel program incurs overheads even when run on a uniprocessor, since it still executes synchronization, parallelism management instructions or partitioning code, or tests to omit these. These overheads can sometimes be significant. The reason that (2) and (3) are different is that the best sequential algorithm may not be possible or easy to parallelize effectively, so the algorithm used in the parallel program may be different than the best sequential algorithm.

Using performance as defined by (3) clearly leads to a better and more honest speedup metric than (1) and (2). From an architect's point of view, however, in many cases it may be okay to use definition (2). Definition (4) fuses uniprocessor performance back into the picture, and thus results in a comparison metric that is similar to absolute performance.

Processing Rate

A metric that is often quoted to characterize the performance of machines is the number of computer operations that they execute per unit time (as opposed to operations that have meaning at application level, such as transactions or chemical bonds). Classic examples are MFLOPS (millions of floating point operations per second) for numerically intensive programs and MIPS (millions of instructions per second) for general programs. Much has been written about why these are not good general metrics for performance, even though they are popular in the marketing literature of vendors. The basic reason is that unless we have an unambiguous, machine-independent measure of the number of FLOPs or instructions needed to solve a problem, these measures can be artificially inflated by using inferior, brute-force algorithms that perform many more FLOPs and take much longer, but produce higher MFLOPS ratings. In fact, we can even inflate the metric by artificially inserting useless but cheap operations that don't access memory). And if the number of operations needed is unambiguously known, then using these rate-based metrics is no different than using execution time. Other problems with MFLOPS, for example, include the fact that different floating point operations have different costs, that even in FLOP-intensive applications modern algorithms use smarter data structures that have many integer operations, and that these metrics are burdened with a legacy of misuse (e.g. for publishing peak hardware performance numbers). While MFLOPS and MIPS are in the best case useful for understanding the basic hardware capabilities, we should be very wary of using them as the main indication of a machine's performance.

Utilization

Architects sometimes measure success by how well (what fraction of the time) they are able to keep their processing engines busy executing instructions rather than stalled due to various overheads. It should be clear by now, however, that processor utilization is not a metric of interest to a

user and not a good sole performance metric at all. It too can be arbitrarily inflated, is biased toward slower processors, and does not say much about end performance or performance bottlenecks. However, it may be useful as a starting point to decide whether to start looking more deeply for performance problems in a program or machine. Similar arguments hold for the utilization of other resources; utilization is useful in determining whether a machine design is balanced among resources and where the bottlenecks are, but not useful for measuring and comparing performance.

Size or Configuration

To compare two systems—or a system with and without a particular enhancement—as the number of processors changes, we could compare performance for a number of well-chosen problem sizes and under different scaling models, using execution time and perhaps speedup as the metrics. However, the results obtained are still dependent on the base problem sizes chosen, and performing the comparison may take a lot of experiments. There is another performance metric that is useful for comparison in some such situations. It is based on the fact that for a given number of processors the inherent overheads of parallelism—inherent communication, load imbalance and synchronization—usually grow relative to useful computation as the problem size is made smaller. From the perspective of evaluating a machine's communication architecture, then, a useful comparison metric as the number of processors increases is the smallest problem size that the machine can run which still achieving a certain measure of parallel "goodness".; for example, the smallest problem that the machine can run with a specified "desirable" or "acceptable" parallel efficiency, where parallel efficiency is defined as speedup divided by the number of processors.[1] By keeping parallel efficiency fixed as the number of processors increases, in a sense this introduces a new scaling model that we might call efficiency-constrained scaling, and with it a performance metric which is the smallest problem size needed. Curves of problem size versus number of processors that maintain a fixed parallel efficiency have been called *isoefficiency* curves [KuG91].

By focusing on the smallest problem sizes that can deliver the desired parallel efficiency, this metric stresses the performance of the mechanisms provided for parallelism, and is particularly useful for comparison when the uniprocessor nodes in the systems being compared are equivalent. However, it may fail when the dominant bottleneck in practice is artifactual communication of a type that increases rather than reduces when smaller problems are run. An example is capacity-induced communication due to nonlocal data in working sets not fitting in the cache. Also, while it stresses communication performance it may not stress the local memory system performance of a processing node. In itself, it does not provide for complete evaluation or comparison of systems. Note that the a similar metric may be useful even when the number of processors is kept fixed and some other parameter like a latency or bandwidth is varied to understand its impact on parallel performance.

---

1. Of course, the notion of a smallest problem is ill-defined when different application parameters can be varied independently by a user, since the same execution time or memory usage may be achieved with different combinations of parameters which have different execution characteristics with regard to parallelism. However, it is uniquely defined given a hard and fast scaling relationship among application parameters (see Section 4.2), or if the user only has direct control over a single parameter and the application automatically scales other parameters appropriately.

In summary, from a user's point of view in comparing machines the *performance* metric of greatest interest is wall-clock execution time (there are of course cost issues to consider). However, from the viewpoint of an architect, a programmer trying to understand a program's performance or a user interested more generally in the performance of a machine, it is best to look at both execution time and speedup. Both these should be presented in the results of any study. Ideally execution time should be broken up into its major components as discussed in Section 3.5. To understand performance bottlenecks, it is very useful to see these component breakdowns on a per-process basis or as some sort of statistic (beyond an average) over processes. In evaluating the impact of changes to the communication architecture, or in comparing parallel machines based on equivalent underlying nodes, size- or configuration-based metrics like the minimum problem size need to achieve a certain goal can be very useful. Metrics like MFLOPS, MIPS and processor utilization can be used for specialized purposes, but using them to truly represent performance requires a lot of assumptions about the knowledge and integrity of the presenter, and they are burdened with a legacy of misuse.

4.3.6 Presenting Results

Given a selected metric(s), there is the issue of how to summarize and finally present the results of a study across a range of applications or workloads The key point here is that although averages and variances over workloads are a concise way to present results, they are not very meaningful particularly for parallel architectures. We have seen the reasons for this: the vast difference among applications with regard to parallel execution characteristics (and even the sharp variations with problem size etc. that we have discussed), the strong dependence of performance on these characteristics, and the lack of a standard accepted suite of workloads that can be considered representative of general-purpose parallel computing. Uniprocessors suffer from some of these problems too, but in much smaller degree. For parallel machines in particular, it is very important to present performance results per application or workload.

An article written in humorous vein [Bai93] and subtitled "Twelve Ways to Fool the Masses" highlights some of these misleading practices that we should guard against. The twelve practices listed are:

Compare 32-bit results to others' 64-bit results The manipulation of 32-bit quantities (such as "single-precision" floating point numbers), is faster than that of 64-bit quantities ("double-precision" floating point), particularly on machines that do not have 64-bit wide data paths.

Present inner kernel results instead of the whole application An inner kernel can be heavily optimized for parallelism or otherwise good performance, but what a user cares about is the performance on the whole application. Amdahl's Law quickly comes into play here.

Use optimized assembly code and compare with others' Fortran or C codes.

Scale problem size with number of processors, but don't tell For example, speedup curves under memory-constrained scaling usually look much better than with a constant problem size.

Quote performance results linearly projected to a full system We could obtain results that show 90% parallel efficiency on a machine with 8 processors (i.e. 7.2-fold speedup), and project that we would also obtain 90% efficiency (57.6-fold speedup) with 64 processors. However, there is little a priori reason to believe that this should be the case.

Compare with scalar, unoptimized, uniprocessor results on Crays At least in high-performance scientific computing, the Cray supercomputers have long been the standard to com-

pare against. We could therefore compare the performance of our 256-processor parallel machine on an application with the same application running non-vectorized on a single processor of the latest parallel Cray machine, and simply say that our machine "beat the Cray".

Compare with old code on an obsolete system Instead of using our new and improved application on a single processor of the latest Cray machine, as above, we could simply use a older dusty-deck version of the application on a much older Cray machine.

Use parallel code as base instead of best sequential one As discussed in Section 4.2, this can inflate speedups.

Quote performance in terms of utilization, speedups, peak MFLOPS per dollar, and leave out execution time See Section 4.3.5.

Use inefficient algorithms to get high MFLOPS rates Again, see Section 4.3.5.

Measure parallel times on a dedicated system, but uniprocessor times on a busy system.

Show pretty pictures and videos, but don't talk about performance.

We should be aware of these ways to mislead both to avoid them ourselves and to avoid being misled by others.

4.4 Evaluating an Architectural Idea or Tradeoff

Imagine that you are an architect at a computer company, getting ready to design your next-generation multiprocessor. You have a new architectural idea that you would like to decide whether or not to include in the machine. You may have a wealth of information about performance and bottlenecks from the previous-generation machine, which in fact may have been what prompted to you to pursue this idea in the first place. However, your idea and this data are not all that is new. Technology and technological assumptions have changed since the last machine, ranging from the level of integration to the cache sizes and organizations used by microprocessors. The processor you will use may be not only a lot faster but also a lot more sophisticated (e.g. four-way issue and dynamically scheduled versus single-issue and statically scheduled), and it may have new capabilities that affect the idea at hand. The operating system has likely changed too, as has the compiler and perhaps even the workloads of interest. And these software components may change further by the time the machine is actually built and sold. The feature you are interested in has a significant cost, in both hardware and particularly design time. And you have deadlines to contend with.

In this sea of change, the relevance of the data that you have in making quantitative decisions about performance and cost is questionable. At best, you could use it together with your intuition to make informed and educated guesses. But if the cost of the feature is high, you would probably want to do more. What you can do is build a simulator that models your system. You fix everything else—the compiler, the operating system, the processor, the technological and architectural parameters—and simulate the system with the feature of interest absent and then present, and judge its performance impact.

Building accurate simulators for parallel systems is difficult. There are many complex interactions in the extended memory hierarchy that are difficult to model correctly, particularly those having to do with contention. Processors themselves are becoming much more complex, and accurate simulation demands that they too be modeled in detail. However, even if you can design

a very accurate simulator that mimics your design, there is still a big problem. Simulation is expensive; it takes a lot of memory and time. The implication for you is that you cannot simulate life-sized problem and machine sizes, and will have to scale down your simulations somehow. This is the first major issue that will be discussed in this section.

Now suppose that your technological parameters are not fixed. You are starting with a clean slate, and want to know how well the idea would work with different technological assumptions. Or you are a researcher, seeking to determine the benefits of an idea in a general context. Now, in addition to the earlier axes of workload, problem size, and machine size, the parameters of the machine are also variable. These parameters including the number of processors, the cache size and organization, the granularities of allocation, communication and coherence, and the performance characteristics of the communication architecture such as latency, occupancy and bandwidth. Together with the parameters of the workload, this leads to a vast parameter space that we must navigate. The high cost and the limitations of simulation make it all the more important that we prune this design space while not losing too much coverage. We will discuss the methodological considerations in choosing parameters and pruning the design space, using a particular evaluation as an example. First, let us take a quick look at multiprocessor simulation.

4.4.1 Multiprocessor Simulation

While multiple processes and processing nodes are simulated, the simulation itself may be run on a uniprocessor workstation. A *reference generator* plays the role of the processors on the parallel machine. It simulates the activities of the processors, and issues memory references (together with a process identifier that tells which processor the reference is from) or commands such as send/receive to a memory system and interconnection network simulator, which represents all the caches and memories in the system (see Figure 4-7). If the simulation is being run on a uniprocessor, the different simulated processes time-share the uniprocessor, scheduled by the reference generator. One example of scheduling may be to deschedule a process every time it issues a reference to the memory system simulator, and allow another process to run until it issues its next reference. Another example would be to reschedule processes every simulated clock cycle. The memory system simulator simulates all the caches and main memories on the different processing nodes, as well as the interconnection network itself. It can be arbitrarily complex in its simulation of data paths, latencies and contention.

The coupling between the reference generator (processor simulator) and the memory system simulator can be organized in various ways, depending on the accuracy needed in the simulation and the complexity of the processor model. One option is trace-driven simulation. In this case, a trace of the instructions executed by each process is obtained by running the parallel program on one system, and the same trace is fed into the simulator that simulates the extended memory hierarchy of a different multiprocessor system. Here, the coupling or flow of information is only in one direction: from the reference generator (here just a trace) to the memory system simulator. the more popular form of simulation is execution-driven simulation, which provides coupling in both directions.

In execution-driven simulation, when the memory system simulator receives a reference or command from the reference generator, it simulates the path of the reference through the extended memory hierarchy—including contention with other references, and returns to the reference generator the time that the reference took to be satisfied. This information, together with concerns about fairness and preserving the semantics of synchronization events, is used by the reference

generator to determine which simulated process to schedule next. Thus, there is feedback from the memory system simulator to the reference generator, providing more accuracy than trace-driven simulation. To allow for maximum concurrency in simulating events and references, most components of the memory system and network are also modeled as separate communicating threads scheduled by the simulator. A global notion of simulated time—i.e. time that would have been seen by the simulated machine, not real time that the simulator itself runs for—is maintained by the simulator. It is what we look up in determining the performance of workloads on the simulated architecture, and it is used to make scheduling decisions. In addition to time, simu-
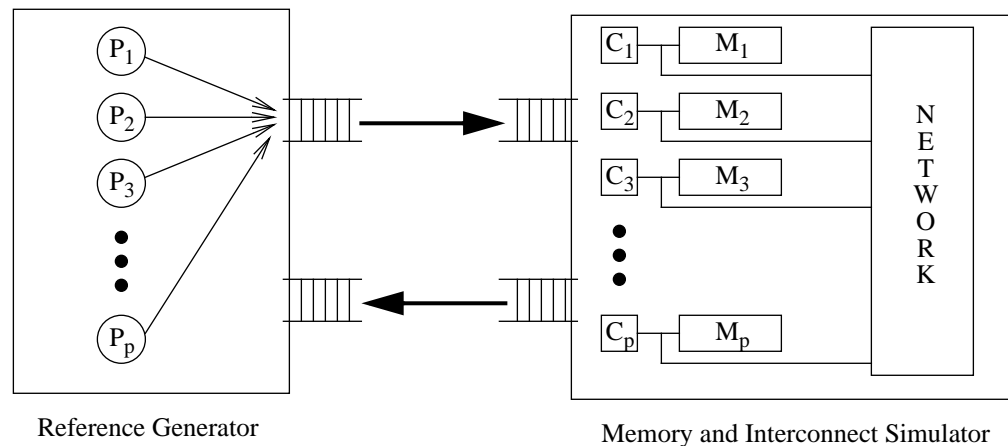
Figure 4-7 Execution-driven multiprocessor simulation.

lators usually keep extensive statistics about the occurrence of various events of interest. This provides us with a wealth of detailed performance information that would be difficult if not impossible to measure on a real system, but some of these types of information may be tainted by a lack of credibility since it is after all simulation. Accurate execution-driven simulation is also much more difficult when complex, dynamically scheduled, multiple-issue processors have to be modeled []. Some of the tradeoffs in simulation techniques are discussed in Exercise 4.8.

Given that the simulation is done in software, and involves many processes or threads that are being very frequently rescheduled (more often for more accuracy) it should not be surprising that simulation is very expensive, and that we need to scale down parameters for it. Research is also being done in performing simulation itself as a parallel application to speed it up, and in using hardware emulation instead of simulation [RHL+93, Gol93, cite Dubois]

4.4.2 Scaling Down Problem and Machine Parameters for Simulation

The tricky part about scaling down problem and machine parameters is that we want the scaled down machine running the smaller problem to be representative of the full-scale machine running the larger problem. Unfortunately, there are no good general formulas for this. But it is an important issue, since it is the reality of most architectural tradeoff evaluation. We should at least understand the limitations of such scaling, recognize which parameters can be scaled down with confidence and which cannot, and develop guidelines that help us avoid pitfalls. Let us first examine scaling down the problem size and number of processors, and then explain some further

difficulties associated with lower-level machine parameters. We focus again on a cache-coherent shared address space communication abstraction for concreteness.

Problem parameters and number of processors

Consider problem parameters first. We should first look for those problem parameters, if any, that affect simulation time greatly but do not have much impact on execution characteristics related to parallel performance. An example is the number of time-steps executed in many scientific computations, or even the number of iterations in the simple equation solver. The data values manipulated can change a lot across time-steps, but the behavioral characteristics often don't change very much. In such cases, after initialization and the cold-start period, which we now ignore, we can run the simulation for a few time-steps and ignore the rest.[1]

Unfortunately, many if not most application parameters affect execution characteristics related to parallelism. When scaling these parameters we must also scale down the number of processors, since otherwise we may obtain highly unrepresentative behavioral characteristics. This is difficult to do in a representative way, because we are faced with many constraints that are individually very difficult to satisfy, and that might be impossible to reconcile. These include:

Preserving the distribution of time spent in program phases The relative amounts of time spent performing different types of computation, for example in different phases of an application, will most likely change with problem and machine size.

Preserving key behavioral characteristics. These include the communication to computation ratio, load balance, and data locality, which may all scale in different ways!

Preserving scaling relationships among application parameters. See Section 4.2.4.

Preserving contention and communication patterns. This is particularly difficult, since burstiness for example is difficult to predict or control.

A more realistic goal than preserving true representativeness when scaling down might be to at least cover a range of realistic operating points with regard to some key behavioral characteristics that matter most for a study, and avoid unrealistic scenarios. In this sense scaled down simulations are not in fact quantitatively representative of any particular reality, but can be used to gain insight and rough estimates. With this more modest and realistic goal, let us assume that we have scaled down application parameters and the number of processors in some way, and see how to scale other machine parameters.

Other machine parameters

Scaled down problem and machine sizes interact differently with low-level machine parameters than the full-scale problems would. We must therefore choose parameters with considerable care.

---

1. We may have to omit the initialization and cold-start periods of the application from the measurements, since their impact is much larger in the run with reduced time-steps than it would be in practice. If we expect that the behavior over long periods of time may change substantially, then we can dump out the program state periodically from an execution on a real machine of the problem configuration we are simulating, and start a few sample simulations with these dumped out states as their input data sets (again not measuring cold-start in each sample). Other sampling techniques can also be used to reduce simulation cost.

Consider the *size of the cache or replication store*. Suppose that the largest problem and machine configuration that we can simulate for the equation solver kernel is a 512-by-512 grid with 16 processors. If we don't scale the 1MB per-processor cache down, we will never be able to represent the situation where the important working set doesn't fit in the cache. The key point with regard to scaling caches is that it should be done based on an understanding of how *working sets* scale and on our discussion of realistic and unrealistic operating points. Not scaling the cache size at all, or simply scaling it down proportionally with data set size or problem size are inappropriate in general, since cache size interacts most closely with working set size, not with data set or problem size. We should also ensure that the caches we use don't become extremely small, since these can suffer from unrepresentative mapping and fragmentation artifacts. Similar arguments apply to other replication stores than processor caches, especially those that hold communicated data.

Example 4-6   In the Barnes-Hut application, suppose that the size of the most important working set when running a full-scale problem with *n*=1M particles is 150KB, that the target machine has 1MB per-processor caches, and that you can only simulate an execution with *n*=16K particles. Would it be appropriate to scale the cache size down proportionally with the data set size? How would you choose the cache size?

Answer   We know from Chapter 3 that the size of the most important working set in Barnes-Hut scales as *log n*, where *n* is the number of particles and is proportional to the size of the data set. The working set of 150KB fits comfortably in the full-scale 1MB cache on the target machine. Given its slow growth rate, this working set is likely to always fit in the cache for realistic problems. If we scale the cache size proportionally to the data set for our simulations, we get a cache size of 1MB * 16K/1M or 16KB. The size of the working set for the scaled down problem is

$150KB * \dfrac{\log 16K}{\log 1M}$ or 70KB, which clearly does not fit in the scaled down 16KB

cache. Thus, this form of scaling has brought us to an operating point that is not representative of reality. We should rather choose a cache size large enough to always hold this working set.

As we move to still lower level parameters of the extended memory hierarchy, including performance parameters of the communication architecture, scaling them representatively becomes increasingly difficult. For example, interactions with cache *associativity* are very difficult to predict, and the best we can do is leave the associativity as it is. The main danger is with retaining a direct mapped cache when cache sizes are scaled down very low, since this is particularly susceptible to mapping conflicts that wouldn't occur in the full-scale cache. Interactions with *organizational parameters of the communication architecture*—such as the granularities of data allocation, transfer and coherence—are also complex and unpredictable unless there is near-perfect spatial locality, but keeping them fixed can lead to serious, unrepresentative artifacts in some cases. We shall see some examples in the Exercises. Finally, *performance parameters* like latency, occupancy, and bandwidth are also very difficult to scale down appropriately with all the others. For example, should the average end-to-end latency of a message be kept the same or reduced according to the number of hops traversed, and how should aggregate bandwidth be treated.
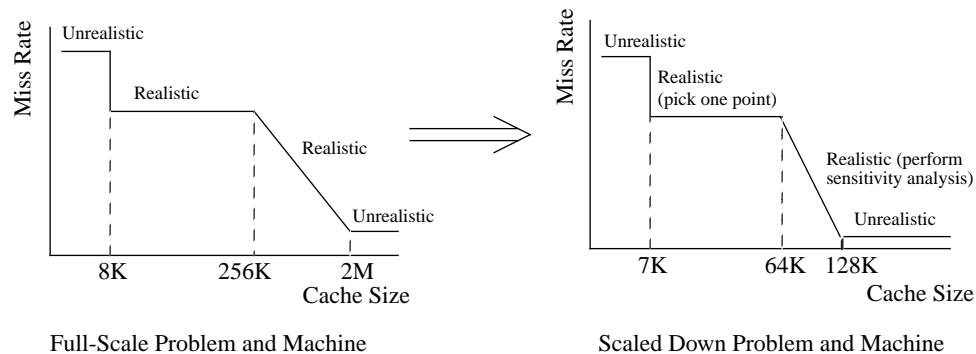
Figure 4-8 Choosing cache sizes for scaled down problem and machines.

Based on our understanding of the sizes and scaling of working sets, we first decide what regions of the curve are realistic for full-scale problems running on the machine with full-scale caches (left graph). We then project or measure what the working set curve looks like for the smaller problem and machine size that we are simulating (right graph), prune out the corresponding unrealistic regions in it, and pick representative operating points (cache sizes) for the realistic regions as discussed in Section 4.4. For regions that cannot be pruned we can perform sensitivity analysis as necessary.

In summary, the best approach to simulation is to try to run the problem sizes of interest to the extent possible. However, by following the guidelines and pitfalls discussed above, we can at least ensure that we cover the important types of operating points with regard to some key execution characteristics when we scale down, and guide our designs by extrapolating with caution. We can be reasonably confident about scaling with respect to some parameters and characteristics, but not others, and our confidence relies on our understanding of the application. This is okay for understanding whether certain architectural features are likely to be beneficial or not, but it is dangerous to use scaled down situations to try to draw precise quantitative conclusions.

4.4.3 Dealing with the Parameter Space: An Example Evaluation

Consider now the problem of the large parameter space opened up by trying to evaluate an idea in a general context, so that lower-level machine parameters are also variable. Let us now examine an actual evaluation that we might perform through simulation, and through it see how we might deal with the parameter space. Assume again a cache-coherent shared address space machine with physically distributed memory. The default mechanism for communication is implicit communication in cache blocks through loads and stores. We saw in Chapter 3 that the impact of end-point communication overhead and network latency can be alleviated by communicating in larger messages. We might therefore wish to understand the utility of adding to such an architecture a facility to explicitly send larger messages, called a block transfer facility, which programs can use in addition to the standard transfer mechanisms for cache blocks. In the equation solver, a process might send an entire border subrow or subcolumn of its partition to its neighbor process in a single block transfer.

In choosing workloads for such an evaluation, we would of course choose at least some with communication that is amenable to being structured in large messages, such as the equation solver. The more difficult problem is navigating the parameter space. Our goals are three-fold:

To avoid unrealistic execution characteristics We should avoid combinations of parameters (or *operating points*) that lead to unrealistic behavioral characteristics; that is, behavior that wouldn't be encountered in practical use of the machine.

To obtain good coverage of execution characteristics We should try to ensure that important characteristics that may arise in real usage are represented.

To prune the parameter space Even in the realistic subspaces of parameter values, we should try to prune out points when possible based on application knowledge, to save time and resources, without losing much coverage, and to determine when explicit sensitivity analysis is necessary.

We can prune the space based on the goals of the study, restrictions on parameters imposed by technology or the use of specific building blocks, and an understanding of parameter interactions. Let us go through the process of choosing parameters, using the equation solver kernel as an example. Although we shall examine the parameters one by one, issues that arise in later stages may make us revisit decisions that were made earlier.

Problem size and number of processors

We choose these based on the considerations of inherent program characteristics that we have discussed in evaluating a real machine and in scaling down for simulation. For example, if the problem is large enough that the communication to computation ratio is very small, then block transfer is not going to help overall performance much. Nor if the problem is small enough that load imbalance is the dominant bottleneck.

What about other architectural parameters? Since problem and machine size as well as these architectural parameters are now variables, we could either fix the latter first and then choose problem size and machine sizes as in Section 4.3, or do the reverse. The reality usually is that we pick one first and then iterate in a brief feedback loop to arrive at a good compromise. Here, we assume that we fix the problem and machine sizes first, primarily because these are limited by simulation resources. Given a problem size and number of processors, let us examine choosing other parameters. For the equation solver, we assume that we are using a 512-by-512 grid and 16 processors.

Cache/Replication size

Once again, we choose cache sizes based on knowledge of the working set curve. In evaluating a real machine we saw how to choose problem sizes given a cache size (and organization) and number of processors. Clearly, if we know the curve and how the important working sets scale with problem size and number of processors, we can keep these fixed and choose cache sizes as well.

Example 4-7   Since the equation solver kernel has sharply defined working sets, it provides a good example for choosing cache sizes for simulation. How might you choose the cache sizes in this case?

Answer   Figure 4-9 illustrates the method using the well-defined working sets of the equation solver. In general, while the sizes of important working sets often depend
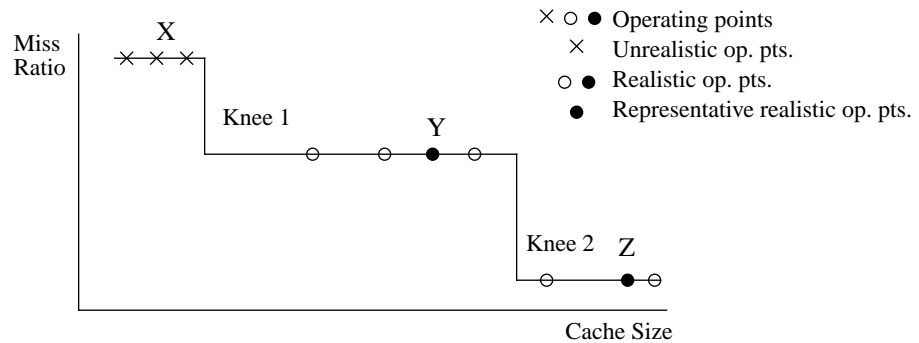
Figure 4-9 Picking cache sizes for an evaluation using the equation solver kernel.

Knee 1 corresponds roughly to a couple of subrows of either B or $n/\sqrt{p}$ elements, depending on whether the grid traversal is blocked or not, and knee 2 to a processor's partition of the matrix, i.e. data set n^2 divided by p. The latter working set may or may not fit in the cache depending on n and p, so both Y and Z are realistic operating points and should be represented. For the first working set, it is conceivable that it not fit in the caches if the traversal is not blocked, but as we have seen in realistically large caches this is very unlikely. If the traversal is blocked, the block size B is chosen so the former working set always fits. Operating point X is therefore representative of an unrealistic region and is ignored. Blocked matrix computations are similar in this respect.

on application parameters and the number of processors, their nature and hence the general shape of the curve usually does not change with these parameters. For an important working set, suppose we know its size and how it scales with these parameters (as we do know for the equation solver, see Figure 4-9). If we also know the range of cache sizes that are realistic for target machines, we can tell whether it is (i) unrealistic to expect that working set to fit in the cache in practical situations, (ii) unrealistic to expect that working set to not fit in the cache, or (iii) realistic to expect it to fit for some practical combinations of parameter values and to not fit for others.[1] Thus, we can tell which of the regions between knees in the curve may be representative of realistic situations and which are not. Given the problem size and number of processors, we know the working set curve versus cache size, and can choose cache sizes to avoid unrepresentative regions, cover representative ones, and prune flat regions by choosing only a single cache size from them (if all we care about from a cache is its miss rate).

Whether or not the important working sets fit in the cache affects the benefits from block transfer greatly, in interesting ways. The effect depends on whether these working sets consist of locally or nonlocally allocated data. If they consist mainly of local data—as in the equation solver when data are placed properly—and if they don't fit in the cache, the processor spends more of its time stalled on the local memory system. As a result, communication time becomes relatively less important and block transfer is likely to help less (block transferred data also interferes more with the local bus traffic, causing contention). However, if the working sets are mostly nonlocal data, we have an opposite effect: If they don't fit in the cache, then there is more communication and

_____

1. Whether a working set of a given size fits in the cache may depend on cache associativity and perhaps even block size in addition to cache size, but it is usually not a major issue in practice if we assume there is at least 2-way associativity as we shall see later, and we can ignore these effects for now.

hence a greater opportunity for block transfer to help performance. Again, proper evaluation requires that we understand the applications well enough.

Of course, a working set curve is not always composed of relatively flat regions separated by sharply defined knees. If there are knees but the regions they separate are not flat (Figure 4-10(a)), we can still prune out entire regions as before if we know them to be unrealistic. However, if a region is realistic but not flat, or if there aren't any knees until the entire data set fits in the cache (Figure 4-10(b)), then we must resort to sensitivity analysis, picking points close to the extremes as well as perhaps some in between.

(a) Portion to left of knee is not flat   (b) No flat regions for realistic caches
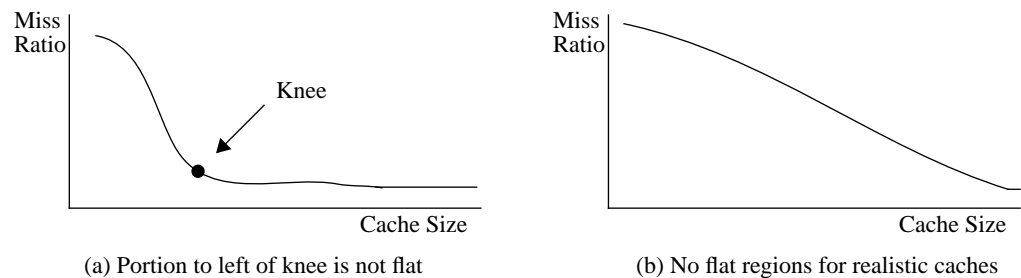
Figure 4-10 Miss rate versus cache size curves that are not knees separated by flat regions.

The remaining question is determining the sizes of the knees and the shape of the curve between them. In simple cases, we may be able to do this analytically. However, complex algorithms and many other factors such as constant factors and the effects of cache block size and associativity may be difficult to analyze. In these cases, we can obtain the curve for a given a problem size and number of processors by measurement (simulation) with different cache sizes. The simulations needed are relatively inexpensive, since working set sizes do not depend on detailed timing-related issues such as latencies, bandwidths, occupancies and contention, which therefore do not need to be simulated. How the working sets change with problem size or number of processors can then be analyzed or measured again as appropriate. Fortunately, analysis of growth rates is usually easier than predicting constant factors. Lower-level issues like block size and associativity often don't change working sets too much for large enough caches and reasonable cache organizations (other than direct-mapped caches) [WOT+95].

Cache block size and associativity

In addition to the problem size, number of processors and cache size, the cache block size is another important parameter for the benefits of block transfer. The issues are a little more detailed. Long cache blocks themselves act like small block transfers for programs with good spatial locality, making explicit block transfer relatively less effective in these cases. On the other hand, if spatial locality is poor then the extra traffic caused by long cache blocks (due to fragmentation or false-sharing) can consume a lot more bandwidth than necessary. Whether this wastes bandwidth for block transfer as well depends on whether block transfer is implemented by pipelining whole cache blocks through the network or only the necessary words. Block transfer itself increases bandwidth requirements, since it causes the same amount of communication to be performed in hopefully less time. If block transfer is implemented with cache block transfers, then with poor spatial locality it may hurt rather than help when available bandwidth is limited, since

it may increase contention for the available bandwidth. Often, we are able to restrict the range of interesting block sizes either due to constraints of current technology or because of limits imposed by the set of building blocks we might use. For example, almost all microprocessors today support cache blocks between 32 and 128 bytes, and we may have already chosen a microprocessor that has a 128-byte cache block. When thresholds occur in the interactions of problem size and cache block size (for instance in the sorting example discussed earlier), we should ensure that we cover both sides of the threshold.

While the magnitude of the impact of cache associativity impact is very difficult to predict, real caches are built with small associativity (usually at most 4-way) so the number of choices to consider is small. If we must choose a single associativity, we are best advised to avoid direct-mapped caches unless we know that the machines of interest will have them.

Performance Parameters of the Communication Architecture

Overhead The higher the overhead of initiating the transfer of a cache block (on a miss, say) the more important it is to amortize it by structuring communication in larger, block transfers. This is true as long as the overhead of initiating a block transfer is not so high as to swamp out the benefits, since the overhead of explicitly initiating a block transfer may be larger than of automatically initiating the transfer of a cache block.

N*etwork transit time* The higher the network transit time between nodes, the greater the benefit of amortizing it over large block transfers (there are limits to this, which will be discussed when we examine block transfer in detail in Chapter 11). The effects of changing latency usually do not exhibit knees or thresholds, so to examine a range of possible latencies we simply have to perform sensitivity analysis by choosing a few points along the range. Usually, we would choose latencies based on the target latencies of the machines of interest; for example, tightly coupled multiprocessors typically have much smaller latencies than workstations on a local area network.

Transfer Bandwidth We have seen that available bandwidth is an important issue for our block transfer study. Bandwidth also exhibits a strong knee effect, which is in fact a saturation effect: Either there is enough bandwidth for the needs of the application, or there is not. And if there is, then it may not matter too much whether the available bandwidth is four times what is needed or ten times. We can therefore pick one bandwidth that is less than that needed and one that is much more. Since the block transfer study is particularly sensitive to bandwidth, we may also choose one that is closer to the borderline. In choosing bandwidths, we should be careful to consider the burstiness in the bandwidth demands of the application; choices based on average bandwidth needs over the whole application may lead to unrepresentative contention effects.

Revisiting choices

Finally, we may often need to revise our earlier choices for parameter values based on interactions with parameters considered later. For example, if we are forced to use small problem sizes due to lack of simulation time or resources, then we may be tempted to choose a very small cache size to represent a realistic case where an important working set does not fit in the cache. However, choosing a very small cache may lead to severe artifacts if we simultaneously choose a direct-mapped cache or a large cache block size (since this will lead to very few blocks in the cache and potentially a lot of fragmentation and mapping conflicts). We should therefore recon-

sider our choice of problem size and number of processors for which we want to represent this situation.

4.4.4 Summary

The above discussion shows that the results of an evaluation study can be very biased and misleading if we don't cover the space adequately: We can easily choose a combination of parameters and workloads that demonstrates good performance benefits from a feature such as block transfer, and can just as easily choose a combination that doesn't. It is therefore very important that we incorporate the above methodological guidelines in our architectural studies, and understand the relevant interactions between hardware and software.

While there are a lot of interactions, we can fortunately identify certain parameters and properties that are high-level enough for us to reason about, that do not depend on lower-level timing details of the machine, and upon which key behavioral characteristics of applications depend crucially. We should ensure that we cover realistic regimes of operation with regard to these parameters and properties. They are: *application parameters*, the *number of processors*, and the *relationship between working sets and cache/replication size* (that is, the question of whether or not the important working sets fit in the caches). Some benchmark suites provide the basic characteristics for their applications, together with their dependence on these parameters, so that architects do not have to reproduce them [WOT+95].

It is also important to look for knees and flat regions in the interactions of application characteristics and architectural parameters, since these are useful for both coverage and pruning. Finally, the high-level goals and constraints of a study can also help us prune the parameter space.

This concludes our discussion of methodological issues in workload-driven evaluation. In the rest of this chapter, we introduce the rest of the parallel programs that we shall use most often as workloads in the book, and describe the basic methodologically relevant characteristics of all these workloads as well as the three we will use that were described in previous chapters (Ocean, Barnes-Hut, and Raytrace).

4.5 Illustrating Workload Characterization

We shall use workloads extensively in this book to quantitatively illustrate some of the architectural tradeoffs we discuss, and to evaluate our case-study machines. Systems that support a coherent shared address space communication abstraction will be discussed in Chapters 5 and 8, while message passing and non-coherent shared address space will be discussed in Chapter 7. Our programs for the abstractions are written in the corresponding programming models. Since programs for the two models are written very differently, as seen in Chapter 2, we illustrate our characterization of workloads with the programs used for a coherent shared address space. In particular, we use six parallel applications and computational kernels that run in batch mode (i.e. one at a time) and do not include operating system activity, and a multiprogrammed workload that includes operating system activity. While the number of applications we use is small, we try to choose applications that represent important classes of computation and have widely varying characteristics.

4.5.1 Workload Case Studies

All of the parallel programs we use for shared address space architectures are taken from the SPLASH2 application suite (see APPENDIX A). Three of them (Ocean, Barnes, and Raytrace) have already been described and used to illustrate performance issues in previous chapters. In this section, we briefly describe the workloads that we will use but haven't yet discussed: LU, Radix (which we have used as an example to illustrate interactions of problem size with cache block size), Radiosity and Multiprog. Of these, LU and Radix are computational kernels, Radiosity is a real application, and Multiprog is a multiprogrammed workload. Then, in Section 4.5.2, we measure some methodologically relevant execution characteristics of the workloads, including the breakdown of memory references, the communication to computation ratio and how it scales, and the size and scaling of the important working sets. We will use this characterization in later chapters to justify our use of default memory system parameters, and to pick the cache sizes that we shall use with these applications and data sets.

LU

Dense *LU* factorization is the process of converting a dense matrix *A* into two matrices *L*, *U* that are lower- and upper-triangular, respectively, and whose product equals *A* (i.e. $A=LU$)[1]. Its utility is in solving linear systems of equations, and it is encountered in scientific applications as well as optimization methods such as linear programming. In addition to its importance, *LU* factorization is a well-structured computational kernel that is nontrivial, yet familiar and fairly easy to understand.

LU factorization works like Gaussian elimination, eliminating one variable at a time by subtracting rows of the matrix by scalar multiples of other rows. The computational complexity of LU factorization is $O(n^3)$ while the size of the data set is $O(n^2)$. As we know from the discussion of temporal locality in Chapter 3, this is an ideal situation to exploit temporal locality by blocking. In fact, we use a blocked LU factorization, which is far more efficient both sequentially and in parallel than an unblocked version. The *n*-by-*n* matrix to be factored is divided into *B*-by-*B* blocks, and the idea is to reuse a block as much as possible before moving on to the next block.

We now think of the matrix as consisting of $\frac{n}{B}$–by–$\frac{n}{B}$ blocks rather than of *n*-by-*n* elements and eliminate and update blocks at a time just as we would elements, but using matrix operations like multiplication and inversion on small B-by-B blocks rather than scalar operations on elements. Sequential pseudocode for this "blocked" LU factorization is shown in Figure 4-11.

Consider the benefits of blocking. If we did not block the computation, a processor would compute an element, then compute its next assigned element to the right, and so forth until the end of the current row, after which it would proceed to the next row. When it returned to the first active element of the next row, it would re-references a perimeter row element (the one it used to compute the corresponding active element of the previous row). However, by this time it has streamed

---

1. A matrix is called dense if a substantial proportion of its elements are non-zero (matrices that have mostly zero entries are called sparse matrices). A lower-triangular matrix such as L is one whose entries are all *0* above the main diagonal, while an upper-triangular matrix such as U has all *0*'s below the main diagonal. The main diagonal is the diagonal that runs from the top left corner of the matrix to the bottom right corner.

```
for k ← 0 to N-1 do                        /*loop over all diagonal blocks */
        factorize block A_k,k;
        for j ← k+1 to N-1 do              /* for all blocks in the row of, and
                                              to the right of, this diagonal block */
                A_k,j ← A_k,j * (A_k,k)^-1;          /* divide by diagonal block */
                for i ← k+1 to N-1 do      /* for all rows below this diagonal block*/
                        for j ← k+1 to N-1 do    /* for all blocks in the corr. row */
                                A_i,j ← A_i,j - A_i,k* (A_k,j)^T;
                        endfor
                endfor
        endfor
endfor
```

Figure 4-11 Pseudocode describing sequential blocked dense *LU* factorization.

N is the number of blocks ($N = n/B$), and $A_{i,j}$ represents the block in the ith row and j^{th} column of matrix A. In the kth iteration of this outer loop, we call the block $A_{k,k}$ on the main diagonal of A the diagonal block, and the kth row and column the perimeter row and perimeter column, respectively (see also Figure 4-12). Note that the kth iteration does not touch any of the elements in the first $k-1$ rows or columns of the matrix; that is, only the shaded part of the matrix in the square region to the right of and below the pivot element is "active" in the current outer loop iteration. The rest of the matrix has already been computed in previous iterations, and will be inactive in this and all subsequent outer loop iterations in the factorization.

through data proportional to an entire row of the matrix, and with large matrices that perimeter row element might no longer be in the cache. In the blocked version, we proceed only B elements in a direction before returning to previously referenced data that can be reused. The operations (matrix multiplications and factorizations) on the B-by-B blocks each involve $O(B^3)$ computation and data accesses, with each block element being accessed B times. If the block size B is chosen such that a block of B-by-B or B^2 elements (plus some other data) fits in the cache, then in a given block computation only the first access to an element misses in the cache. Subsequent accesses hit in the cache, resulting in B^2 misses for B^3 accesses or a miss rate of $\frac{1}{B}$.

Figure 4-12 provides a pictorial depiction of the flow of information among blocks within an outer loop iteration, and also shows how we assign blocks to processors in the parallel version. Because of the nature of the computation, blocks toward the top left of the matrix are active only in the first few outer loop iterations of the computation, while blocks toward the bottom right have a lot more work associated with them. Assigning contiguous rows or squares of blocks to processes (a domain decomposition of the matrix) would therefore lead to poor load balance. We therefore interleave blocks among processes in both dimensions, leading to a partitioning called a two-dimensional *scatter decomposition* of the matrix: The processes are viewed as forming a two-dimensional $\sqrt{p}$-by-$\sqrt{p}$ grid, and this grid of processes is repeatedly stamped over the matrix of blocks like a cookie cutter. A process is responsible for computing the blocks that are assigned to it in this way, and only it writes to those blocks. The interleaving improves—but does not eliminate—load balance, while the blocking preserves locality and allows us to use larger data transfers on message passing systems.

The drawback of using blocks rather than individual elements is that it increases task granularity (the decomposition is into blocks rather than elements) and hurts load balance: Concurrency is reduced since there are fewer blocks than elements, and the maximum load imbalance per itera-

for all k from 0 to N-1

   if I own A_{kk}, factorize A_{kk}

   BARRIER;

   for all my blocks A_{kj} in pivot ro

      $A_{kj} \leftarrow A_{kj} * A_{kk}^{-1}$

   BARRIER;

   for all my blocks Aij in active i

      $A_{ij} \leftarrow A_{ij} - A_{ik} * A_{kj}$

endfor

Diagonal Block

Perimeter Block

Interior Block

"Cookie Cutter"
assignment of
blocks to processors
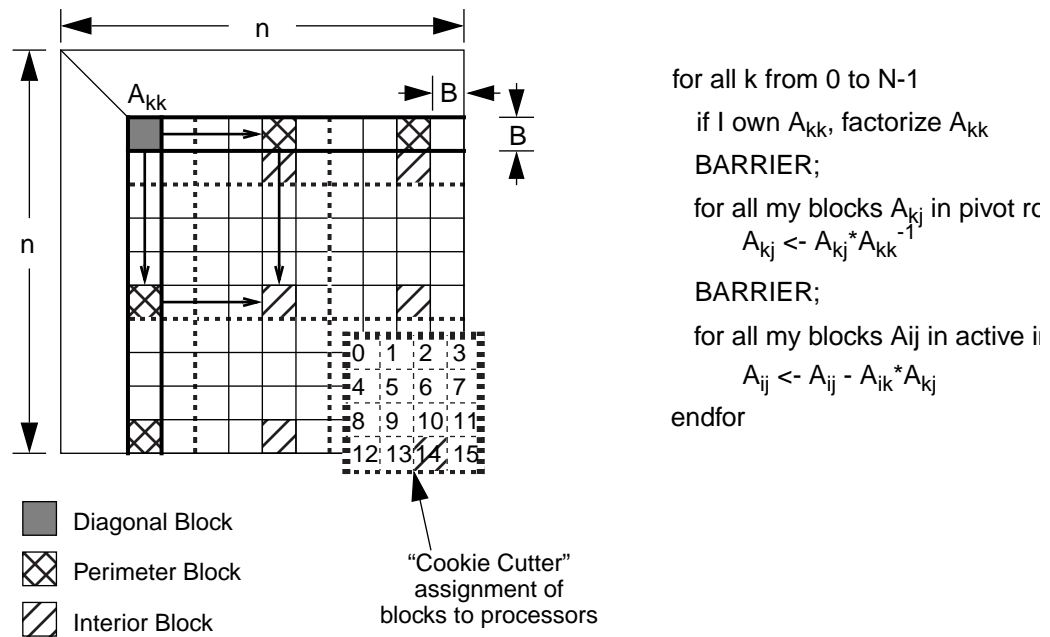
Figure 4-12 Parallel blocked LU factorization: flow of information, partitioning and parallel pseudocode.

The flow of information within an outer (*k*) loop iteration is shown by the solid arrows. Information (data) flows from the diagonal block (which is first factorized) to all blocks in the perimeter row in the first phase. In the second phase, a block in the active part of the matrix needs the corresponding elements from the perimeter row and perimeter column.

tion is the work associated with a block rather than a single element. In sequential *LU* factorization, the only constraint on block size is that the two or three blocks used in a block computation fit in the cache. In the parallel case, the ideal block size *B* is determined by a tradeoff between data locality and communication overhead (particularly on a message-passing machine) pushing toward larger blocks on one hand and load balance pushing toward smaller blocks on the other. The ideal block size therefore depends on the problem size, number of processors, and other architectural parameters. In practice block sizes of 16-by-16 or 32-by-32 elements appear to work well on large parallel machines.

Data can also be reused across different block computations. To reuse data from remote blocks, we can either copy blocks explicitly in main memory and keep them around or, on cache-coherent machines perhaps rely on caches being large enough to do this automatically. However, reuse across block computations is typically not nearly as important to performance as reuse within them, so we do not make explicit copies of blocks in main memory.

For spatial locality, since the unit of decomposition is now a two-dimensional block, the issues are quite similar to those discussed for the simple equation solver kernel in Section 3.4.1. We are therefore led to a three- or four-dimensional array data structure to represent the matrix in a shared address space. This allows us to distribute data among memories at page granularity (if blocks are smaller than a page, we can use one more dimension to ensure that all the blocks assigned to a process are contiguous in the address space). However, with blocking the capacity miss rate is small enough that data distribution in main memory is often not a major problem,

unlike in the equation solver kernel. The more important reason to use high-dimensional arrays is to reduce cache mapping conflicts across subrows of a block as well as across blocks, as we shall see in Chapter 5 (Section 5.7). These are very sensitive to the size of the array and number of processors, and can easily negate most of the benefits of blocking.

No locks are used in this computation. Barriers are used to separate outer loop iterations as well as phases within an iteration (e.g. to ensure that the perimeter row is computed before the blocks in it are used). Point-to-point synchronization at the block level could have been used instead, exploiting more concurrency, but barriers are much easier to program.

Radix

The Radix program sorts a series of integers, called *keys*, using the radix sorting method. Suppose there are n integers to be sorted, each of size b bits. The algorithm uses a radix of r bits, where r is chosen by the user. This means the b bits representing a key can be viewed as a set of $\lceil b/r \rceil$ groups of r bits each. The algorithm proceeds in $\lceil b/r \rceil$ phases or iterations, each phase sorting the keys according to their values in the corresponding group of r bits, starting with the lowest-order group (see Figure 4-13).[1] The keys are completely sorted at the end of these $\lceil b/r \rceil$
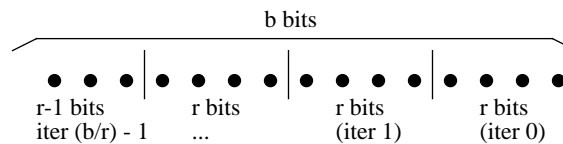
Figure 4-13 A b-bit number (key) divided into $\lceil b/r \rceil$ groups of r bits each.

The first iteration of radix sorting uses the least significant r bits, etc.

phases. Two one-dimensional arrays of size n integers are used: one stores the keys as they appear in the input to a phase, and the other the output from the phase. The input array for one phase is the output array for the next phase, and vice versa.

The parallel algorithm partitions the n keys in each array among the p processes so that process *0* get the first *n/p* keys, processor *1* the next *n/p* keys, and so on. The portion of each array assigned to a process is allocated in the corresponding processor's local memory (when memory is physically distributed). The *n/p* keys in the input array for a phase that are assigned to a process are called its local keys for that phase. Within a phase, a process performs the following steps:

1. Make a pass over the local *n/p* keys to build a local (per-process) histogram of key values. The histogram has 2^r entries. If a key encountered has the value i in the r bits corresponding to the current phase, then the ith bin of the histogram is incremented.

2. When all processes have completed the previous step (determined by barrier synchronization in this program), accumulate the local histograms into a global histogram. This is done with a

---

1. The reason for starting with the lowest-order group of r bits rather than the highest-order one is that this leads to a "stable" sort; i.e. keys with the same value appear in the output in the same order relative to one another as they appeared in the input.

parallel prefix computation, as discussed in Exercise 4.13. The global histogram keeps track of both how many keys there are of each value, and also for each of the p process-id values j, how many keys there are of a given value that are owned by processes whose id is less than j.

3. Make another pass over the local n/p keys. For each key, use the global and local histograms to determine which position in the output array this key should go to, and write the key value into that entry of the output array. Note that the array element that will be written is very likely (expected likelihood $(p-1)/p$) to be nonlocal (see Figure 4-14). This phase is called the *permutation* phase.
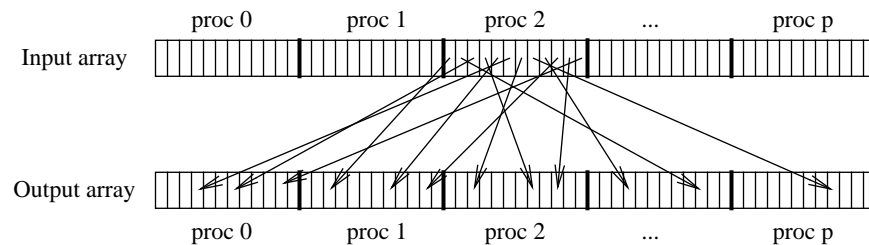
Figure 4-14 The permutation phase of a radix sorting iteration.

In each of the input and output arrays (which change places in successive iterations), keys (entries) assigned to a process are allocated in the corresponding processor's local memory.

A more detailed description of the algorithm and how it works can be found in [BL+91]. In a shared address space implementation, communication occurs when writing the keys in the permutation phase (or reading them in the binning phase of the next iteration if they stay in the writers' caches), and in constructing the global histogram from the local histograms. The permutation-related communication is all-to-all personalized (i.e. every process sends some keys to every other) but is irregular and scattered, with the exact patterns depending on the distribution of keys and the iteration. The synchronization includes global barriers between phases, and finer-grained synchronization in the phase that builds the global histogram. The latter may take the form of either mutual exclusion or point-to-point event synchronization, depending on the implementation of this phase (see Exercises).

Radiosity

The radiosity method is used in computer graphics to compute the global illumination in a scene containing diffusely reflecting surfaces. In the hierarchical radiosity method, a scene is initially modeled as consisting of k large input polygons or *patches*. Light transport interactions are computed pairwise among these patches. If the light transfer between a pair of patches is larger than a threshold, then one of them (the larger one, say) is subdivided, and interactions computed between the resulting patches and the other patch. This process continues until the light transfer between all pairs is low enough. Thus, patches are hierarchically subdivided as necessary to improve the accuracy of computing illumination. Each subdivision results in four subpatches, leading to a quadtree per patch. If the resulting final number of undivided subpatches is n, then with k original patches the complexity of this algorithm is $O(n+k^2)$. A brief description of the steps in the algorithm follows. Details can be found in [HSA91, Sin93].

The input patches that comprise the scene are first inserted into a binary space partitioning (BSP) tree [FAG83] to facilitate efficient visibility computation between pairs of patches. Every input patch is initially given an *interaction list* of other input patches which are potentially visible from it, and with which it must therefore compute interactions. Then, radiosities are computed by the following iterative algorithm:

1. For every input patch, compute its radiosity due to all patches on its interaction list, subdividing it or other patches hierarchically and computing their interactions recursively as necessary (see below and Figure 4-15).

2. Starting from the patches at the leaves of the quadtrees, add all the area-weighted patch radiosities together to obtain the total radiosity of the scene, and compare it with that of the previous iteration to check for convergence within a fixed tolerance. If the radiosity has not converged, return to step 1. Otherwise, go to step 3.

3. Smooth the solution for display.

Most of the time in an iteration is spent in step 1. Suppose a patch i is traversing its interaction
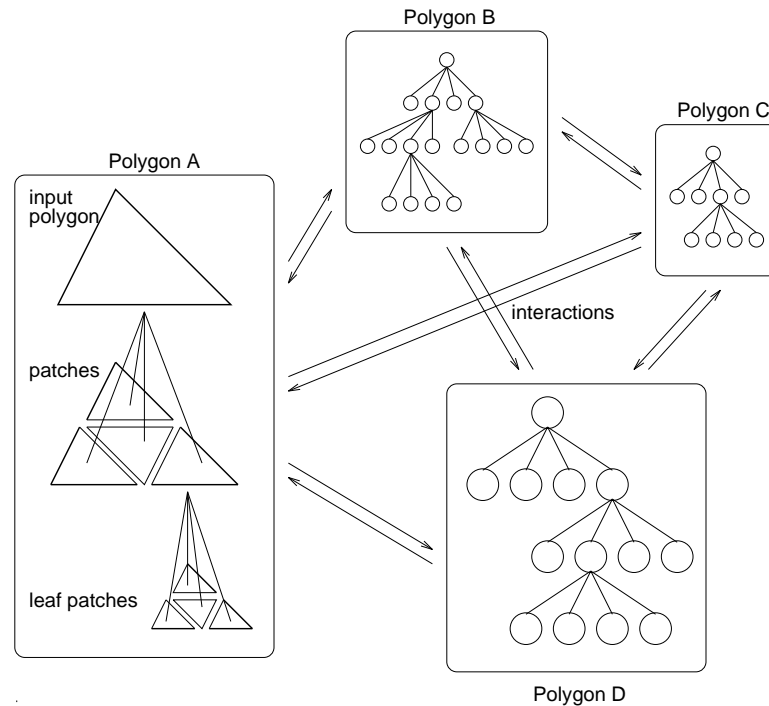
Figure 4-15 Hierarchical subdivision of input polygons into quadtrees as the radiosity computation progresses.

Binary trees are shown instead of quadtrees for clarity, and only one patch's interactions lists are shown.

list to compute interactions with other patches (quadtree nodes). Consider an interaction between patch i and another patch j on its interaction list. The interaction involves computing the inter-visibility of the two patches and the light transfer between them (the actual light transfer is the product of the actual inter-visibility and the light transfer that would have happened assuming full inter-visibility). Computing inter-visibility involves traversing the BSP tree several times from one patch to the other[1]; in fact, visibility computation is a very large portion of the overall execu-

tion time. If the result of an interaction says that the "source" patch i should be subdivided, then four children are created for patch i—if they don't already exist due to a previous interaction—and patch j is removed from i's interaction list and added to each of i's children's interaction lists, so that those interactions can be computed later. If the result is that patch j should be subdivided, then patch j is replaced by its children on patch i's interaction list. This means that interactions will next be computed between patch i and each of patch j's children. These interactions may themselves cause subdivisions, so the process continues recursively (i.e.if patch j's children are further subdivided in the course of computing these interactions, patch i ends up computing interactions with a tree of patches below patch j; since the four children patches from a subdivision replace the parent in-place on i's interaction list, the traversal of the tree comprising patch j's descendants is depth-first). Patch i's interaction list is traversed fully in this way before moving on to the next patch (perhaps a descendant of patch i, perhaps a different patch) and its interaction list. Figure 4-16 shows an example of this hierarchical refinement of interactions. The next itera-
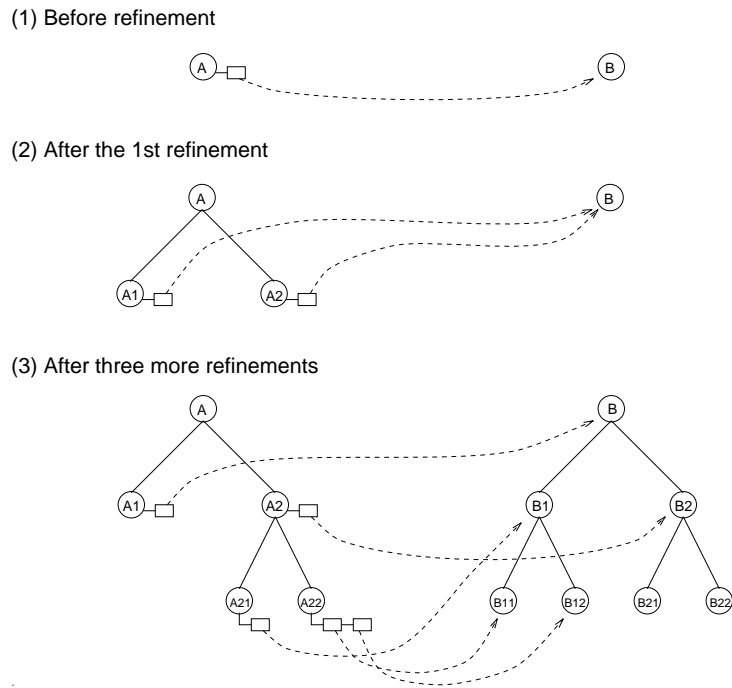
Figure 4-16 Hierarchical refinement of interactions and interaction lists.

tion of the iterative algorithm starts with the quadtrees and interaction lists as they are at the end of the current iteration.

---

1. Visibility is computed by conceptually shooting a number of rays between the two patches, and seeing how many of the rays reach the destination patch without being occluded by intervening patches in the scene. For each such conceptual ray, determining whether it is occluded or not is done efficiently by traversing the BSP tree from the source to the destination patch.

Parallelism is available at three levels in this application: across the *k* input polygons, across the patches that these polygons are subdivided into—i.e. the patches in the quadtrees—and across the interactions computed for a patch. All three levels involve communication and synchronization among processors. We obtain the best performance by defining a task to be either a patch and all its interactions or a single patch-patch interaction, depending on the size of the problem and the number of processors.

Since the computation and subdivisions are highly unpredictable, we have to use task queues with task stealing. The parallel implementation provides every processor with its own task queue. A processor's task queue is initialized with a subset of the initial polygon-polygon interactions. When a patch is subdivided, new tasks involving the subpatches are enqueued on the task queue of the processor that did the subdivision. A processor processes tasks from its queue until there are no tasks left. Then, it steals tasks from other processors' queues. Locks are used to protect the task queues and to provide mutually exclusive access to patches as they are subdivided (note that to different patches, assigned to two different processes, may have the same patch on their interaction list, so both processes may try to subdivide the latter patch at the same time). Barrier synchronization is used between phases. The parallel algorithm is non-deterministic, and has highly unstructured and unpredictable execution characteristics and access patterns.

Multiprog

The workloads we have discussed so far include only parallel application programs running one at a time. As we said at the beginning of the previous chapter, a very common use of multiprocessors, particularly small-scale shared-address-space multiprocessors, is as throughput engines for multiprogrammed workloads. The fine-grained resource sharing supported by these machines allows a single operating system image to service the multiple processors efficiently. Operating system activity is often itself a substantial component of such workloads, and the operating system constitutes an important, complex parallel application in itself. The final workload we study is a multiprogrammed (time-shared) workload, consisting of a number of sequential applications and the operating system itself. The applications are two UNIX file `compress` jobs, and two parallel compilations—or `pmakes`—in which multiple files needed to create an executable are compiled in parallel. The operating system is a version of UNIX produced by Silicon Graphics Inc., called IRIX version 5.2.

4.5.2 Workload Characteristics

We now quantitatively measure some important basic characteristics of all our workloads, including the breakdown of memory references, the communication to computation ratio and how it scales, and the size and scaling of the important working sets. We use this characterization to justify our choice of memory system parameters, and to pick the cache sizes that we shall use with these applications and data sets in the rest of our evaluations (according to the methodology developed in this chapter). We present data for 16-processor executions for our parallel applications, and 8-processor executions of the multiprogrammed workload. For these simulations we assume that each processor has a single-level, 1 Mbyte large, 4-way set-associative cache with 64-byte cache blocks.

Data Access and Synchronization Characteristics

Table 4-1 summarizes the basic reference counts and dynamic frequency of synchronization events (locks and global barriers) in the different workloads. From left-to-right the columns are: program name, input data-set, total instructions executed, total number of floating-point operations executed, reads, writes, shared-reads, shared-writes, barriers, lock-unlock pairs. A dash in a table entry means that the entry is not applicable for that application (e.g. Radix has no floating point operations). The input data sets are the default data sets that we shall use for these programs in the simulation experiments in the rest of the book, unless otherwise mentioned. The data sets are chosen to be large enough to be of practical interest for a machine of about 64 processors (the size we will measure in Chapter 8), but small enough to simulate in reasonable time. They are at the small end of the data sets one might run in practice on 64-processor machines, but are quite appropriate for the bus-based machines we consider here. How the characteristics of interest scale with problem size is usually discussed qualitatively or analytically, and sometimes measured.

In all the programs throughout the book, we begin keeping track of behavioral and timing statistics after the child processes are created by the parent. Previous references (by the main process) are issued to the memory system simulator and simulated, but are not included in the statistics. In most of the applications, measurement begins exactly after the child processes are created. The exceptions are Ocean and Barnes-Hut. In both these cases we are able to take advantage of the fact that we can drastically reduce the number of time-steps for the purpose of simulation (as discussed in Section 4.4.2) but have to then ignore cold-start misses and allow the application to settle down before starting measurement. We simulate a small number of time-steps—six for Ocean and five for Barnes-Hut—and start tracking behavioral and timing statistics after the first two time-steps (though we do simulate the first two as well). For the Multiprog workload, statistics are gathered from a checkpoint taken close to the beginning of the pmake. While for all other applications we consider only application data references, for the Multiprog workload we also consider impact of instruction references, and furthermore separate out kernel references and user application references. The statistics are presented separately for kernel and user references, and in some cases these are further split into instruction and data references.

As per our discussion in this chapter, we quantitatively characterize the following basic properties of the program here: concurrency and load balance (to show that the programs are appropriate for the machine sizes we with to examine), inherent communication to computation ratio, and working sets. We also discuss analytically or qualitatively how these characteristics scale with key problem parameters and the number of processors. More detailed architectural measurements will be made in later chapters.

Concurrency and Load Balance

We characterize load balance by measuring the algorithmic speedups we discussed earlier; i.e. speedups on an a PRAM, an architectural model that assumes that memory references have zero latency (they just cost the instruction it takes to issue the reference) regardless of whether they are satisfied in the cache, local memory or have to traverse the network. On such an architecture, which is of course impossible to realize in practice, deviations from ideal speedup are attributable to load imbalance, serialization due to critical sections, and the overheads of redundant computation and parallelism management.
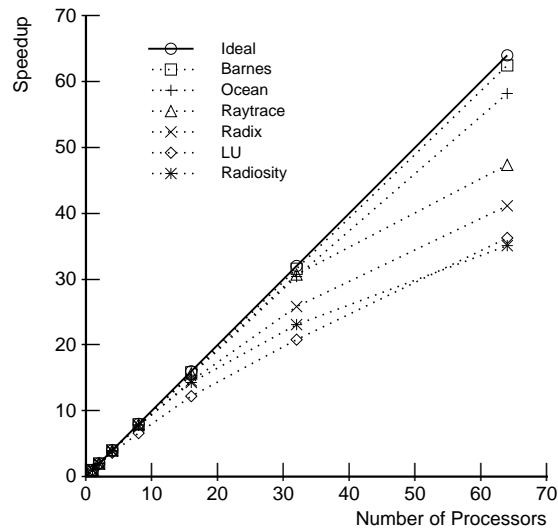
Figure 4-17 PRAM speedups for the six parallel applications.

Ideal speedup simply means a speedup of *p* with *p* processors.

Table 4-1 General statistics about application programs. For the parallel programs, shared reads and writes simply refer to all non-stack references issued by the application processes. All such references do not necessarily point to data that are truly shared by multiple processes. The Multiprog workload is not a parallel application, so does not access shared data. A dash in a table entry means that measurement is not applicable to or is not measured for that application.

Application	Data Set	Total Instr (M)	Total Flops (M)	Total Refs (M)	Total Reads (M)	Total Writes (M)	Shared Reads (M)	Shared Writes (M)	Bar-riers	Locks
LU	512x512 matrix 16x16 blocks	489.52	92.20	151.07	103.09	47.99	92.79	44.74	66	0
Ocean	258x258 grids tolerance=10^{-7} 4 time-steps	376.51	101.54	99.70	81.16	18.54	76.95	16.97	364	1296
Barnes	16K particles θ = 1.0 3 time-steps	2002.74	239.24	720.13	406.84	313.29	225.04	93.23	7	34516
Radix	256K points radix = 1024	84.62	—	14.19	7.81	6.38	3.61	2.18	11	16
RayTrace	Car scene	833.35	—	290.35	210.03	80.31	161.10	22.35	0	94456
Radiosity	Room scene	2297.19	—	769.56	486.84	282.72	249.67	21.88	10	210485
Multiprog: User	SGI Irix 5.2, two pmakes + two compress jobs	1296.43	—	500.22	350.42	149.80	—	—	—	—
Multiprog: Kernel		668.10	—	212.58	178.14	34.44	—	—	—	621505

Figure 4-17 shows the PRAM speedups for the programs for up to 64 processors with the default data sets. Three of the programs (Barnes, Ocean and Raytrace) speed up very well all the way to 64 processors even with the relatively small default data sets. The dominant phases in these programs are data parallel across a large data set (all the particles in Barnes, an entire grid in Ocean, and the image plane in Raytrace), and have limited parallelism and serialization only in some global reduction operations and in portions of some particular phases that are not dominant in terms of number of instructions executed (e.g. tree building and near the root of the upward pass in Barnes, and the higher levels of the multigrid hierarchy in Ocean).

The programs that do not speed up fully well for the higher numbers of processors with these data sets are LU, Radiosity, and Radix. The reasons in all these cases have to do with the sizes of the input data-sets rather than the inherent nature of load imbalance in the applications themselves. In LU, the default data set results in considerable load imbalance for 64 processors, despite the block-oriented decomposition. Larger data sets reduce the imbalance by providing more blocks per processor in each step of the factorization. For Radiosity, the imbalance is also due to the use of a small data set, though it is very difficult to analyze. Finally, for Radix the poor speedup at 64 processors is due to the prefix computation when accumulating local histograms into a global histogram (see Section), which cannot be completely parallelized. The time spent in this prefix computation is $O(\log p)$ while the time spent in the other phases is $O(n/p)$, so the fraction of total work in this unbalanced phase decreases as the number of keys being sorted increases. Thus, even these three programs can be used to evaluate larger machines as long as larger data sets are chosen.

We have therefore satisfied our first criterion of not choosing parallel programs that are inherently unsuitable for the machine sizes we want to evaluate, and of understanding how to choose data sets for these programs that make them appropriate in this regard. Let us now examine the inherent communication to computation ratios and working set sizes of the programs.

Communication to Computation Ratio

We include in the communication to computation ratio inherent communication as well as communication due to the first time a word is accessed by a processor, if it happens to be nonlocally allocated (cold start misses). Where possible, data is distributed appropriately among physically distributed memories, so we can consider this cold start communication to be quite fundamental. To avoid artifactual communication due to finite capacity or issues related to spatial locality (see Section 3.4.1), we simulate infinite per-processor caches and a single-word cache block. We measure communication to computation ratio as the number of bytes transferred per instruction, averaged over all processors. For floating-point intensive applications (LU and Ocean) we use bytes per FLOP (floating point operation) instead of per instruction, since FLOPs are less sensitive to vagaries of the compiler than instructions.

We first present the communication to computation (measured as indicated above) for the base problem size shown in Table 4-1 versus the number of processors used. This shows how the ratio increases with the number of processors under constant problem size scaling. Then, where possible we describe analytically how the ratio depends on the data set size and the number of processors (Table 4-2). The effects of other application parameters on the communication to computation ratio are discussed qualitatively.

Table 4-2 Growth Rates of Communication-to-Computation Ratio.

Code	Growth Rate of Comm/Comp Ratio
LU	$\sqrt{P} / \sqrt{DS}$
Ocean	$\sqrt{P} / \sqrt{DS}$
Barnes	approximately $\sqrt{P} / \sqrt{DS}$
Radiosity	unpredictable
Radix	(P-1)/P
Raytrace	unpredictable

Figure 4-18 shows the first set of results for our six parallel programs. The programs for which
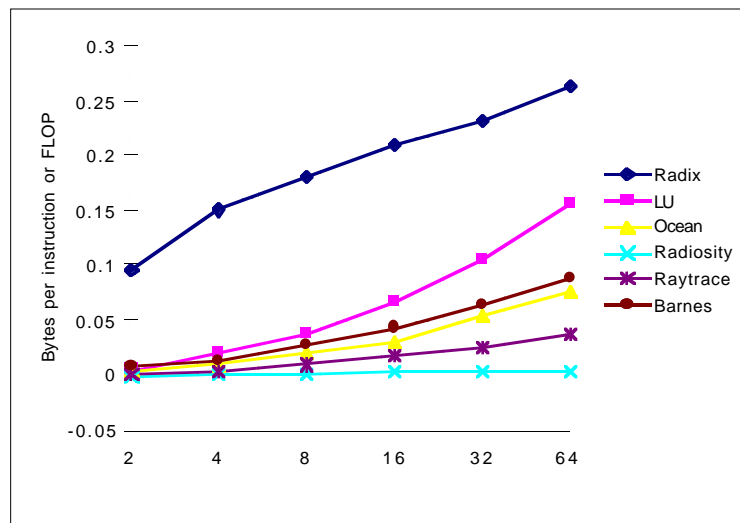
Figure 4-18 Communication to computation ratio versus processor count for the base problem size in the six parallel applications

we measure bytes per instruction are shown on a different graph than the ones for which we measure bytes per FLOP, and Radix is shown in a separate graph due to the need for a different scale on the Y-axis. The first thing we notice is that for the communication to computation ratios are generally quite small. With processors operating at 200 million instructions per second (MIPS), a ratio of 0.1 byte per instruction is about 20MB/sec of traffic. Actual traffic is much higher than inherent—both due to artifactual communication and also because control information is sent along with data in each transfer—and this communication is averaged over the entire execution of the program, but it still quite small for modern, high-performance multiprocessor networks. The only application for which the ratio becomes quite high is Radix, so for this application communication bandwidth is very important to model carefully in evaluations. One reason for the low communication to computation ratios is that the applications we are using have been very well optimized in their assignments for parallel execution. Applications that run on real machines in reality will likely fall more toward higher communication to computation ratios.

The next observation we can make from the figure is that the growth rates of communication to computation ratio are very different across applications. These growth rates with the number of processors and with data set size (not shown in the figures) are summarized in Table 4-2. The

absolute numbers would change dramatically if we used a different data set size (problem size) in some applications (e.g. Ocean and LU), but at least inherent communication would not change much in others. Artifactual communication is a whole different story, and we shall examine communication traffic due to it in the context of different architectural types in later chapters.

While growth rates are clearly fundamental, it is important to realize that growth rates in themselves do not reveal the constant factors in the expressions for communication to computation ratio, which can often be more important than growth rates in practice. For example, if a program's ratio increases only as the logarithm of the number of processors, or almost not at all as in Radix, this does not mean that its ratio is small. In fact, although its ratio will asymptotically become smaller than that of an application whose ratio grows as the square root of the number of processors, it may actually be much larger for all practical machine sizes if its constant factors are much larger. The constant factors for our applications can be determined from the absolute values of the communication to computation ratio in the results we present.

Working Set Sizes

The inherent working set sizes of a program are best measured using fully associative caches and a one-word cache block, and simulating the program with different cache sizes to find knees in the miss rate versus cache size curve. Ideally, the cache sizes should be changed with very fine granularity in the vicinity of knees in order to identify the locations of knees precisely. Finite associativity can make the size of cache needed to hold the working set larger than the inherent working set size, as can the use of multi-word cache blocks (due to fragmentation). We come close to measuring inherent working set sizes by using one-level fully associative caches per processor, with a least-recently-used (LRU) cache replacement policy and 8-byte cache blocks. We use caches sizes that are powers of two, and change cache sizes at a finer granularity in areas where the change in miss rate with cache size is substantial.

Figure 4-19 shows the resulting miss rate versus cache size curves for our six parallel applications, with the working sets labeled as level 1 working set (lev1WS), level 2 working set (lev2WS), etc. In addition to these working sets, some of the applications also have tiny working sets that do not scale with problem size or the number of processors and are therefore always expected to fit in a cache; we call these the level 0 working sets (lev0WS). They typically consist of stack data that are used by the program as temporary storage for a given primitive calculation (such as a particle-cell interaction in Barnes-Hut) and reused across these. These are marked on the graphs when they are visible, but we do not discuss them further.

We see that in most cases the working sets are very sharply defined. Table 4-3 summarizes how the different working sets scale with application parameters and the number of processors, whether they are important to performance (at least on efficient cache-coherent machines), and whether they can be expected to not fit in a modern secondary cache for realistic problem sizes (with a reasonable degree of cache associativity, at least beyond direct-mapped). The cases in which it is reasonable to expect a working set that is important to performance on an efficient cache-coherent machine to not fit in a modern secondary cache are those with a "Yes" in each of the last two columns of the table; i.e. Ocean, Radix and Raytrace. Recall that in Ocean, all the major computations operate on (and stream through) a process's partition of one or more grids. In the near-neighbor computations, fitting a subrow of a partition in the cache allows us to reuse that subrow, but the large working set consists of a process's partitions of entire grids that it might benefit from reusing. Whether or not this large working set fits in a modern secondary cache

(a) LU   (b) Ocean   (c) Barnes

(d) Radiosity   (e) Raytrace   (f) Radix
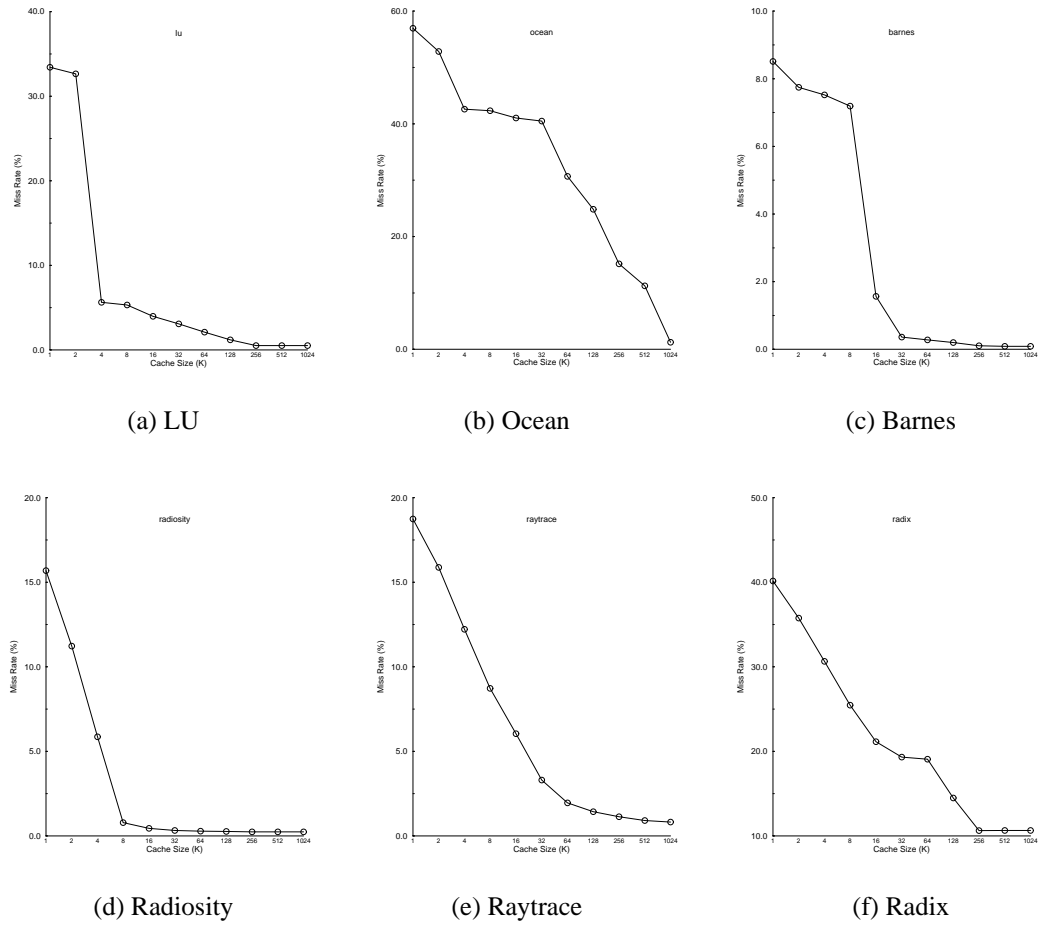
Figure 4-19 Working set curves for the six parallel applications in sixteen-processor executions.

The graphs show miss rate versus cache size for fully associative first-level caches per processor and an 8-byte cache block.

Table 4-3 Important working sets and their growth rates for the SPLASH-2 suite. *DS* represents the data set size, and *P* is the number of processes.

Program	Working Set 1	Growth Rate	Important?	Realistic to not fit in cache?	Working Set 2	Growth Rate	Important?	Realistic to not fit in cache?
LU	One block	Fixed(B)	Yes	No	partition of DS	DS/P	No	Yes
Ocean	A few subrows	$\sqrt{P}/\sqrt{DS}$	Yes	No	partition of DS	DS/P	Yes	Yes
Barnes	Tree data for 1 body	$(\log DS)/\theta^2$	Yes	No	partition of DS	DS/P	No	Yes
Radiosity	BSP tree	$\log(polygons)$	Yes	No	Unstructured	Unstructured	No	Yes
Radix	Histogram	Radix r	Yes	No	partition of DS	DS/P	Yes	Yes
Raytrace	Unstructured	Unstructured	Yes	No	Unstructured	Unstructured	Yes	Yes

therefore depends on the grid size and the number of processors. In Radix, a process streams through all its n/p keys, at the same time heavily accessing the histogram data structure (of size proportional to the radix used). Fitting the histogram in cache is therefore important, bit this working set is not clearly defined since the keys are being streamed through at the same time. The larger working set is when a process's entire partition of the key data set fits in the cache, which may or may not happen depending on *n* and *p*. partitions that a process might reuse. Finally, in Raytrace we have seen that the working set is diffuse and ill-defined, and can become quite large depending on the characteristics of the scene being traced and the viewpoint. For the other applications, we expect the important working sets to fit in the cache for realistic problem and machine sizes. We shall take this into account as per our methodology when we evaluate architectural tradeoffs in the following chapters. In particular, for Ocean, Radix and Raytrace we shall choose cache sizes that both fit and do not fit the larger working set, since both situations are representative of practice.

4.6 Concluding Remarks

We now have a good understanding of the major issues in workload-driven evaluation for multiprocessors: choosing workloads, scaling problems and machines, dealing with the large parameter space, and presenting results. For each issue, we have a set of guidelines and steps to follow, an understanding of how to avoid pitfalls, and a means to understand the limitations of our own and other investigations. We also have a basis for our own quantitative illustration and evaluation of architectural tradeoffs in the rest of the book. However, our experiments will mostly serve to illustrate important points rather than evaluate tradeoffs comprehensively, since the latter would require a much wider range of workloads and parameter variations.

We've seen that workloads should be chosen to represent a wide range of applications, behavioral patterns, and levels of optimization. While complete applications and perhaps multiprogrammed workloads are indispensable, there is a role for simpler workloads such as microbenchmarks and kernels as well.

We have also seen that proper workload-driven evaluation requires an understanding of the relevant behavioral properties of the workloads and their interactions with architectural parameters. While this problem is complex, we have provided guidelines for dealing with the large parameters space—for evaluating both real machines as well as architectural tradeoffs—and pruning it while still obtaining coverage of realistic situations.

The importance of understanding relevant properties of workloads was underscored by the scaling issue. Both execution time and memory may be constraints on scaling, and applications often have more than one parameter that determines execution properties. We should scale programs based on an understanding of these parameters, their relationships, and their impact on execution time and memory requirements. We saw that realistic scaling models driven by the needs of applications lead to very different results of architectural significance than naive models that scale only a single application parameter.

Many interesting issues also arose in our discussion of choosing metrics to evaluate the goodness of a system or idea and presenting the results of a study. For example, we saw that execution time (preferably with a breakdown into its major components) and speedup are both very useful met-

rics to present, while rate-based metrics such as MFLOPS or MIPS or utilization metrics are too susceptible to problems.

Finally, with this chapter we have described all of the workloads that we shall use in our own illustrative workload-driven evaluation of real systems and architectural tradeoffs in the rest of the book, and have quantified their basic characteristics. We are now on a firm footing to use them as we proceed to examine detailed architectural issues in the remainder of the book.

4.7 References

[Bai93]   D. H. Bailey, Misleading Performance Reporting in the Supercomputing Field, Scientific Programming, vol. 1., no. 2, p. 141 - 151, 1993. Early version in Proceedings of Supercomputing'93.

[Bai91]   D. H. Bailey, Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers, Supercomputing Review, August 1991, p. 54 - 55.

[Par94]   ParkBench Committee. Public International Benchmarks for Parallel Computers. Scientific Programming, vol. 3, no. 2 (Summer 1994).

[Bai90]   D.H. Bailey, FFTs in External or Hierarchical Memory, Journal of Supercomputing, vol. 4, no. 1 (March 1990), p. 23 - 35. Early version in Proceedings of Supercomputing'89, Nov. 1989, p. 234-242.

[BBB+91]   D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. The NAS Parallel Benchmarks, Intl. Journal of Supercomputer Applications, vol. 5, no. 3 (Fall 1991), pp. 66 - 73. Also Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility. NASA Ames Research Center March 1994.

[BHS+95]   David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility. NASA Ames Research Center December, 1995

[BH89]   Barnes, J. E. and Hut, P. Error Analysis of a Tree Code. Astrophysics Journal Supplement, 70, pp. 389-417, June 1989.

[GBJ+94]   A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. PVM: A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.

[Sun90]   V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. Concurrency: Practice and Experience, vol. 2, no. 4, pp 315--339, December, 1990.

[SDG+94]   V. Sunderam, J. Dongarra, A. Geist, and R Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. Parallel Computing, Vol. 20, No. 4, April 1994, pp 531-547.

[GS92]   Network Based Concurrent Computing on the PVM System, G. A. Geist and V. S. Sunderam, Journal of Concurrency: Practice and Experience, 4, 4, pp 293--311, June, 1992.

[MPI94]   The Message Passing Interface, International Journal of Supercomputing Applications Volume 8 Number 3/4, Fall/Winter 1994 (updated 5/95). Special issue on MPI.

[SOL+95]   Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference. MIT Press, 1995.

[BC+89]   Berry, M., Chen, D. et al. The PERFECT Club Benchmarks: Effective Performance Evaluation of Computers. International Journal of Supercomputer Applications, 3(3), pp. 5-40, 1989.

[BL+91]   Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J, and Zagha, M. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Sympo-*

sium on Parallel Algorithms and Architectures, pp. 3-16, July 1991.

[Do90]   Dongarra, Jack J. Performance of Various Computers using Standard Linear Equations Software in a Fortran Environment. Technical Report CS-89-85, Computer Science Department, University of Tennessee, March 1990.

[DW93]   Dongarra, Jack J. and Gentzsch, W (editors). Computer Benchmarks. Elsevier Science B.V., North-Holland, Amsterdam, 1993.

[DMW87]   Dongarra, Jack J., Martin, J., and Worlton, J. Computer Benchmarking: Paths and Pitfalls. IEEE Spectrum, July 1987, p. 38.

[DW94]   Dongarra, Jack J. and Walker, David W. Software libraries for linear algebra computations on high performance computers, SIAM Review, 37 (1995), pp. 151-180.

[CDP+92]   J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia, 1992, IEEE Computer Society Press, pp. 120-127.

[BCC+97]   L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley. ScaLAPACK Users' Guide. SIAM Press, July 1997.

[KLS+94]   C. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, The High Performance Fortran Handbook. MIT Press, 1994.

[FoW78]   S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In Proceedings of the Tenth ACM Symposium on Theory of Computing, May 1978.

[FAG83]   Fuchs, H., Abram, G. and Grant, E. Near real-time shaded display of rigid objects. In Proceedings of SIGGRAPH, 1983.

[Gol93]   Goldschmidt, S.R. Simulation of Multiprocessors: Speed and Accuracy, Ph.D. Dissertation, Stanford University, June 1993 <<tech report info>>.

[Gus88]   Gustafson, J.L. Reevaluating Amdahl's Law. *Communications of the ACM*, vol 31, no. 5, May 1988, pp. 532-533.

[GS94]   Gustafson, J.L and Snell, Q.O. HINT: A New Way to Measure Computer Performance. Technical Report, Ames Laboratory, U.S. Department of Energy, Ames, Iowa, 1994. <<number>>

[HSA91]   Hanrahan, P., Salzman, D. and Aupperle, L. A Rapid Hierarchical Radiosity Algorithm. In Proceedings of SIGGRAPH, 1991.

[Her87]   Hernquist, L. Performance Characteristics of Tree Codes. Astrophysics Journal Supplement, 64, pp. 715-734, August 1987.

[Hey91]   Hey, A.J.G. The Genesis Distributed Memory Benchmarks. Parallel Computing, 17, pp. 1111-1130, 1991.

[HPF93]   High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, vol. 2, no. 1, 1993, p. 170.

[KuG91]   Vipin Kumar and Anshul Gupta. Analysis of Scalability of Parallel Algorithms and Architectures: A Survey. In Proceedings of the International Conference on Supercomputing, pp. 396-405, 1991.

[LO+87]   Lusk, E.W., Overbeek, R. et al. Portable Programs for Parallel Processors. Holt, Rinehart and Winston, Inc. 1987.

[PAR94]   PARKBENCH Committee. Public International Benchmarks for Parallel Computers. Scientific Computing, 1994 <<more info>>. Also Technical Report CS93-213, Department of Computer Science, University of Tennessee, Knoxville, November 1993.

[RH+94]   Rosenblum, M., Herrod, S. A., Witchel, E. and Gupta, A. Complete Computer Simulation: The

SimOS Approach. IEEE Parallel and Distributed Technology, vol. 3, no. 4, Fall 1995.

[RSG93]   Rothberg, E., Singh, J.P., and Gupta, A. Working Sets, Cache Sizes and Node Granularity for Large-scale Multiprocessors. In Proc. Twentieth International Symposium on Computer Architecture, pp. 14-25, May 1993.

[SGC93]   Saavedra, R. H., Gaines, R.S., and Carlton, M.J. Micro Benchmark Analysis of the KSR1. In Proceedings of Supercomputing'93, November 1993, pp. 202-213.

[Sin93]   Singh, J.P. Parallel Hierarchical N-body methods and their Implications for Multiprocessors. Ph.D. Thesis, Technical Report no. CSL-TR-93-565, Stanford University, March 1993.

[SGL94]   Singh, J.P., Gupta, A. and Levoy, M. Parallel Visualization Algorithms: Performance and Architectural Implications. IEEE Computer, vol. 27, no. 6, June 1994.

[SH+95]   Singh, J.P., Holt, C., Totsuka, T., Gupta, A. and Hennessy, J.L. Load balancing and data locality in Hierarchical N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. Journal of Parallel and Distributed Computing, 1995 <<complete citation>>.

[SWG92]   Singh, J.P., Weber, W.-D., and Gupta, A., SPLASH: The Stanford Parallel Applications for Shared Memory, Computer Architecture News, 20(1), March 1992, pp. 5-44.

[SPE89]   Standard Performance Evaluation Corporation. SPEC Benchmark Suite Release 1.0, 1989.

[WH95]   Wood, D.A. and Hill, M.D. Cost-Effective Parallel Computing. IEEE Computer, vol. 28, no. 2, February 1995, pp. 69-72.

4.8 Exercises

4.1 Design a few simple exercises.

4.2 <4.2> Scaling

    a.   What is the fundamental problem with TC scaling? Illustrate it with an example.

    b.   Suppose you had to evaluate the scalability of a system. One possibility is to measure the speedups under different scaling models as defined in this chapter. Another is to see the how the problem size needed to get say 70% parallel efficiency scales. What are the advantages and particularly disadvantages or caveats for each of these? What would you actually do?

4.3 <4.3.5> Your boss asks you to compare two types of systems based on same uniprocessor node, but with some interesting differences in their communication architectures. She tells you that she cares about only 10 particular applications. She orders you to come up with a single numeric measure of which is better, given a fixed number of processors and a fixed problem size for each application, despite your arguments based on reading this chapter that averaging over parallel applications is not such a good idea. What additional questions would you ask her before running the applications with those problem sizes on the systems. What measure of average would you report to her and why (explain in some detail).

4.4 <4.3.5> Often, a system may display good speedups on an application even though its communication architecture is not well suited to the application. Why might this happen? Can you design a metric alternative to speedup that measures the effectiveness of the communication architecture for the application? Discuss some of the issues and alternatives in designing such a metric.

4.5 <4.3> A research paper you read proposes a communication architecture mechanism and tells you that starting from a given communication architecture on a machine with 32 processors, the mechanism improves performance by 40% on some workloads that are of interest to you.

Is this enough information for you to decide to include that mechanism in the next machine you design? If not, list the major reasons why, and say what other information you would need. Assume that the machine you are to design also has 32 processors.

4.6 <4.3> Suppose you had to design experiments to compare different methods for implementing locks on a shared address space machine. What performance properties would you want to measure, and what "microbenchmark" experiments would you design? What would be your specific performance metrics? Now answer the same questions for global barriers.

4.7 <4.3> You have designed a method for supporting a shared address space communication abstraction transparently in software across bus-based shared memory multiprocessors like the Intel Pentium Quads discussed in Chapter 1. With a node or Quad, coherent shared memory is supported with high efficiency in hardware; across nodes, it is supported much less efficiently and in software. Given a set of applications and the problem sizes of interest, you are about to write a research report evaluating your system. What are the interesting performance comparisons you might want to perform to understand the effectiveness of your cross-node architecture? What experiments would you design, what would each type of experiment tell you, and what metrics would you use? Your system prototype has 16 bus-based multiprocessor nodes with 4 processors in each, for a total of 64 processors.

4.8 <4.4.1> Two types of simulations are often used in practice to study architectural tradeoffs: trace-driven and execution-driven. In trace-driven simulation, a trace of the instructions executed by each process is obtained by running the parallel program on one system, and the same trace is fed into the simulator that simulates the extended memory hierarchy of a different multiprocessor system. In execution-driven simulation, the instruction generator and the memory system simulator are coupled together in both directions. The instruction generator generates a memory instruction from a process; that instruction is simulated by the memory system simulator, and the time at which the operation completes is fed back to the instruction generator, which uses this timing information to determine which process to issue an instruction from next and when to issue the next instruction from a particular process.

a. What are the major tradeoffs between trace-driven and instruction-driven simulation? Under what conditions do you expect the results (say a program's execution time) to be significantly different.

b. What aspects of a system do you think it is most difficult to simulate accurately (processor, memory system, network, communication assist, latency, bandwidth, contention)? Which of these do you think are most important, and which would you compromise on?

c. How important do you think it is to simulate the processor pipeline appropriately when trying to evaluate the impact of tradeoffs in the communication architecture. For example, while many modern processors are superscalar and dynamically scheduled, a single-issue statically scheduled processor is much easier to simulate. Suppose the real processor you want to model is 200MHz with 2-way issue but achieved a perfect memory CPI of 1.5. Could you model it as a single-issue 300MHz processor for a study that wants to understand the impact of changing network transit latency on end performance? What are the major issues to consider?

4.9 <4.4> You are the evaluation specialist for your company. The head of engineering comes to you with a proposal for an enhancement to the communication architecture, and wants you to evaluate its performance benefits for three applications—Radix, Barnes-Hut, and Ocean—using your simulation technology. No further guidelines are given. Your simulation technology is capable of simulating up to 256 processors and up to 8M keys for Radix, up to 32K bodies for Barnes-Hut, and up to a 2050-by-2050 grid size for Ocean. Design a set of experi-

ments that you would report results for to your boss, as well as the manner in which you would report the results.

4.10 <4.4> Consider the familiar iterative nearest-neighbor grid computation on a two-dimensional grid, with subblock partitioning. Suppose we use a four-dimensional array representation, where the first two dimensions are pointers to the appropriate partition. The full-scale problem we are trying to evaluate is a 8192-by-8192 grid of double-precision elements with 256 processors, with 256KB of direct-mapped cache with a 128B cache block size per processor. We cannot simulate this, but instead simulate a 512-buy-512 grid problem with 64 processors.

 a. What cache sizes would you choose, and why?

 b. List some of the dangers of choosing too small a cache.

 c. What cache block size and associativity would you choose, and what are the issues and caveats involved?

 d. To what extent would you consider the results representative of the full-scale problem on the larger machine? Would you use this setup to evaluate the benefits of a certain communication architecture optimization? To evaluate the speedups achievable on the machine for that application?

4.11 <4.2.3, 4.2.4> In scientific applications like the Barnes-Hut galaxy simulation, a key issue that affects scaling is error. These applications often simulate physical phenomena that occur in nature, using several approximations to represent a continuous phenomenon by a discrete model and solve it using numerical approximation techniques. Several application parameters represent distinct sources of approximation and hence of error in the simulation. For example, in Barnes-Hut the number of particles n represents the accuracy with which the galaxy is sampled (spatial discretization), the time-step interval Δt represents the approximation made in discretizing time, and the force calculation accuracy parameter θ determines the approximation in that calculation. The goal of an application scientist in running larger problems is usually to reduce the overall error in the simulation and have it more accurately reflect the phenomenon being simulated. While there are no universal rules for how scientists will scale different approximations, a principle that has both intuitive appeal and widespread practical applicability for physical simulations is the following: *All sources of error should be scaled so that their error contributions are about equal.*

For the Barnes-Hut galaxy simulation, studies in astrophysics [Her87,BH89] show that while some error contributions are not completely independent, the following rules emerge as being valid in interesting parameter ranges:

- n: An increase in n by a factor of s leads to a decrease in simulation error by a factor of $\sqrt{s}$.

- Δt: The method used to integrate the particle orbits over time has a global error of the order of Δt^2. Thus, reducing the error by a factor of $\sqrt{s}$ (to match that due to a s-fold increase in n) requires a decrease in Δt by a factor of $\sqrt[4]{s}$. This means $\sqrt[4]{s}$ more time-steps to simulate a fixed amount of physical time, which we assume is held constant.

- θ: The force-calculation error is proportional to θ^2 in the range of practical interest. Reducing the error by a factor of $\sqrt{s}$ thus requires a decrease in θ by a factor of $\sqrt[4]{s}$.

 Assume throughout this exercise that the execution time of a problem size on p processors is $1/p$ of the execution time on one processor; i.e. perfect speedup for that problem size under problem-constrained scaling.

a. How would you scale the other parameters, θ and Δt if n is increased by a factor of s? Call this rule *realistic* scaling, as opposed to *naive* scaling, which scales only the number of particles n.

b. In a sequential program, the data set size is proportional to n and independent of the other parameters. Do the memory requirements grow differently under realistic and naive scaling in a shared address space (assume that the working set fits in the cache)? In message passing?

c. The sequential execution time grows roughly as $\frac{1}{\Delta t} \cdot \frac{1}{\theta^2} \cdot n\log n$, (assuming that a fixed amount of physical time is simulated). If n is scaled by a factor of s, how does the parallel execution time on p processors scale under realistic and under naive scaling, assuming perfect speedup?

d. How does the parallel execution time grow under MC scaling, both naive and realistic, when the number of processors increases by a factor of k? If the problem took a day on the base machine (before it was scaled up), how long will it take in the scaled up case on the bigger machine under both the naive and realistic models?

e. How does the number of particles that can be simulated in a shared address space grow under TC scaling, both realistic and naive, when the number of processors increases by a factor of k?

f. Which scaling model appears more practical for this application: MC or TC?

4.12 <4.2.3, 4.2.4> For the Barnes-Hut example, how do the following execution characteristics scale under realistic and naive MC, TC and PC scaling?

a. The communication to computation ratio. Assume first that it depends only on n and the number of processors p, varying as $\sqrt{p}/\sqrt{n}$, and roughly plot the curves of growth rate of this ratio with the number of processors under the different models. Then comments on the effects of the other parameters under the different scaling models.

b. The sizes of the different working sets, and hence the cache size you think is needed for good performance on a shared address space machine. Roughly plot the growth rate for the most important working set with number of processors under the different models. What major methodological conclusion in scaling does this reinforce? Comment on any differences in these trends and the amount of local replication needed in the locally essential trees version of message passing.

c. The frequency of synchronization (per unit computation, say), both locks and barriers. Describe qualitatively at least.

d. The average frequency and size of input/output operations, assuming that every processor prints out the positions of all its assigned bodies (i) every ten time-steps, (ii) every fixed amount of physical time simulated (e.g. every year of simulated time in the galaxy's evolution).

e. The number of processors likely to share (access) a given piece of body data during force calculation in a coherent shared address space at a time. As we will see in Chapter 8, this information is useful in the design of cache coherence protocols for scalable shared address space machines.

f. The frequency and size of messages in an explicit message passing implementation, focusing on the communication needed for force calculation and assuming that each processor sends only one message to each other communicating the data that the latter needs from the former to compute its forces.

4.13 <4.5> The radix sorting application required a parallel prefix computation to compute the global histogram from local histograms. A simplified version of the computation is as follows. Suppose each of the *p* processes has a *local value* it has computed (think of this as representing the number of keys for a given digit value in the local histogram of that process). The goal is to compute an array of *p* entries, in which entry *i* is the sum of all the local values from processors 0 through *i-1*. Describe and implement the simplest linear method to compute this output array. Then design a parallel method with a shorter critical path [Hint: you can use a tree structure]. Implement the latter method, and compare its performance with that of the simple linear method on a machine of your choice. You may use the simplified example here, or the fuller example where the "local value" is in fact an array with one entry per radix digit, and the output array is two-dimensional, indexed by process identifier and radix digit. That is, the fuller example does the computation for each radix digit rather than just one.

CHAPTER 5   Shared Memory Multiprocessors

5.1 Introduction

The most prevalent form of parallel architecture is the multiprocessor of moderate scale that provides a global physical address space and symmetric access to all of main memory from any processor, often called a Symmetric Multiprocessor or SMP. Every processor has its own cache adn all the processors and memory modules attach to the same interconnect, usually a shared bus. SMPs dominate the server market and are becoming more common on the desktop. They are also important building blocks for larger scale systems. The efficient sharing of resources, such as memory and processors, makes these machines attractive as "throughput engines" for multiple sequential jobs with varying memory and CPU requirements, and the ability to access all shared data efficiently using ordinary loads and stores from any of the processors makes them attractive for parallel programming. The automatic movement and replication of shared data in the local caches can significantly ease the programming task. These features are also very useful for the

operating system, whose different processes share data structures and can easily run on different processors.

From the viewpoint of the layers of the communication architecture in Figure 5-1, the hardware directly supports the shared address space programming model. User processes can read and write shared virtual addresses, and these operations are realized by individual loads and stores of shared physical addresses that are uniformly and efficiently accessible from any processor. In fact, the relationship between the programming model and the hardware operation is so close, that they both are often referred to simply as "shared memory." A message passing programming model can be supported by an intervening software layer—typically a runtime library—that manages portions of the shared address space explicitly as message buffers per process. A send-receive operation pair is realized by copying data between buffers. The operating system need not be involved, since address translation and protection on these shared buffers is provided by the hardware. For portability, most message passing programming interfaces have been implemented on popular SMPs, as well as traditional message passing machines. In fact, such implementations often deliver higher message passing performance than traditional message passing systems—as long as contention for the shared bus and memory does not become a bottleneck—largely because of the lack of operating system involvement in communication. The operating system is still there for input/output and multiprogramming support.
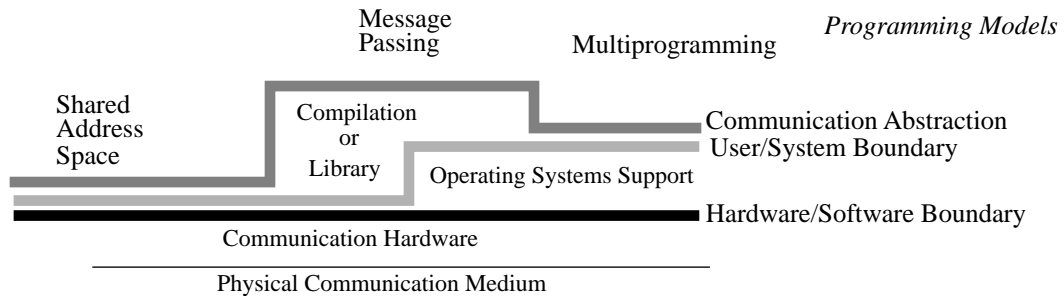
Figure 5-1 Layers of abstraction of the communication architecture for bus-based SMPs.

Since all communication and local computation generates memory accesses in a shared address space, from a system architect's perspective the key high-level design issue is the organization of the extended memory hierarchy. In general, memory hierarchies in multiprocessors fall primarily into four categories, as shown in Figure 5-2, which correspond loosely to the scale of the multi-processor being considered. The first three are symmetric multiprocessors (all of main memory is equally far away from all processors), while the fourth is not.

In the first category, the "shared cache" approach (Figure 5-2a), the interconnect is located between the processors and a shared first-level cache, which in turn connects to a shared main-memory subsystem. Both the cache and the main-memory system may be interleaved to increase available bandwidth. This approach has been used for connecting very small numbers (2-8) of processors. In the mid 80s it was a common technique to connect a couple of processors on a board; today, it is a possible strategy for a multiprocessor-on-a-chip, where a small number of processors on the same chip share an on-chip first-level cache. However, it applies only at a very small scale, both because the interconnect between the processors and the shared first level cache
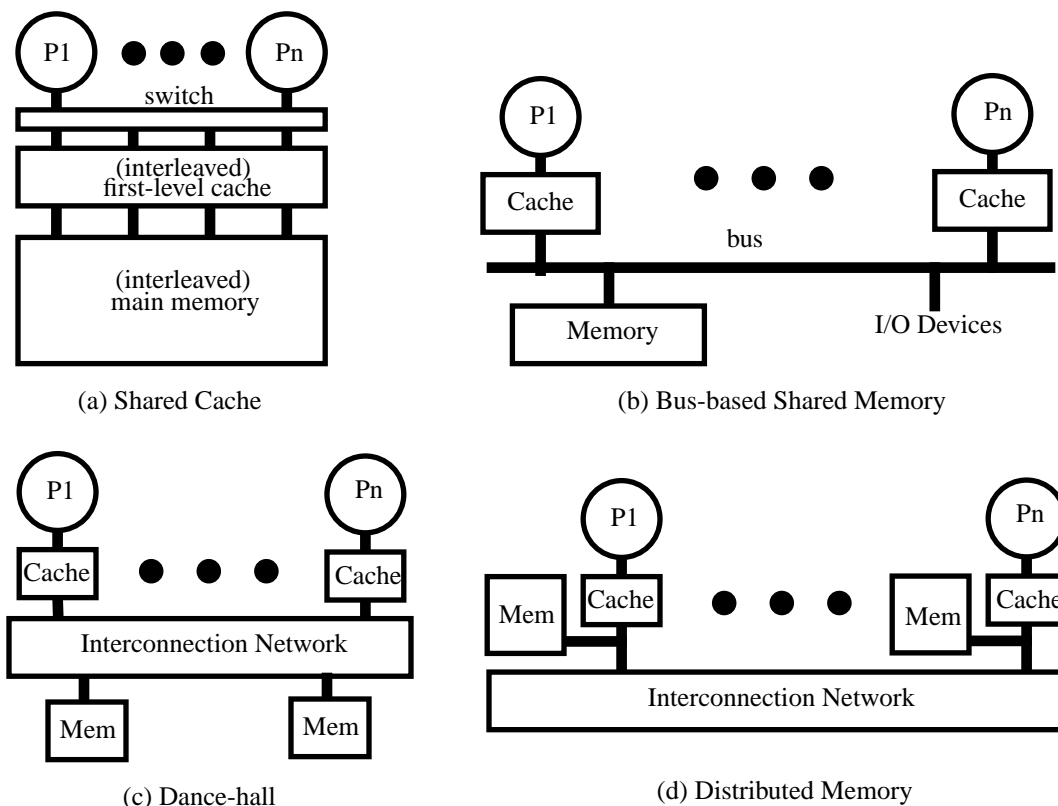
Figure 5-2 Common memory hierarchies found in multiprocessors.

is on the critical path that determines the latency of cache access and because the shared cache must deliver tremendous bandwidth to the processors pounding on it.

In the second category, the "bus-based shared memory" approach (Figure 5-2b), the interconnect is a shared bus located between the processor's private caches and the shared main-memory subsystem. This approach has been widely used for small- to medium-scale multiprocessors consisting of up to twenty or thirty processors. It is the dominant form of parallel machine sold today, and essentially all modern microprocessors have invested considerable design effort to be able to support "cache-coherent" shared memory configurations. For example, the Intel Pentium Pro processor can attach to a coherent shared bus without any glue logic, and low-cost bus-based machines that use these processors have greatly increased the popularity of this approach. The scaling limit for these machines comes primarily due to bandwidth limitations of the shared bus and memory system.

The last two categories are intended to be scalable to many processing nodes. The third category, the "dance-hall" approach, also places the interconnect between the caches and main memory, but the interconnect is now a scalable point-to-point network and memory is divided into many logical modules that connect to logically different points in the interconnect. This approach is symmetric, all of main memory is uniformly far away from all processors, but its limitation is that all of memory is indeed *far* away from all processors: Especially in large systems, several

"hops" or switches in the interconnect must be traversed to reach any memory module. The fourth category, the "distributed memory" approach, is not symmetric: A scalable interconnect is located between processing nodes but each node has its local portion of the global main memory to which it has faster access (Figure 5-2c). By exploiting locality in the distribution of data, the hope is that most accesses will be satisfied in the local memory and will not have to traverse the network. This design is therefore most attractive for scalable multiprocessors, and several chapters are devoted to the topic later in the book. Of course, it is also possible to combine multiple approaches into a single machine design—for example a distributed memory machine whose individual nodes are bus-based SMPs, or sharing a cache other than at the first level.

In all cases, caches play an essential role in reducing the average memory access time as seen by the processor and in reducing the bandwidth requirement each processor places on the shared interconnect and memory system. The bandwidth requirement is reduced because the data accesses issued by a processor that are satisfied in the cache do not have to appear on the bus; otherwise, every access from every processor would appear on the bus which would quickly become saturated. In all but the shared cache approach, each processor has at least one level of its cache hierarchy that is private and this raises a critical challenge, namely that of *cache coherence*. The problem arises when a memory block is present in the caches of one or more processors, and another processor modifies that memory block. Unless special action is taken, the former processors will continue to access the old, stale copy of the block that is in their caches.

Currently, most small-scale multiprocessors use a shared bus interconnect with per-processor caches and a centralized main memory, while scalable systems use physically distributed main memory. Dancehall and shared cache approaches are employed in relatively specific settings, that change as technology evolves. This chapter focuses on the logical design of multiprocessors that exploit the fundamental properties of a bus to solve the cache coherence problem. The next chapter expands on the hardware design issues associated with realizing these cache coherence techniques. The basic design of scalable distributed-memory multiprocessors will be addressed in Chapter 7, and issues specific to scalable cache coherence in Chapter 8.

In Section 5.2 describes the cache coherence problem for shared-memory architectures in detail. Coherence is a key hardware design concept and is a necessary part of our intuitive notion the memory abstraction. However, parallel software often makes stronger assumptions about how memory behaves. Section 5.3 extends the discussion of ordering begun in Chapter 1 and introduces the concept of memory consistency which defines the semantics of shared address space. This issue has become increasingly important in computer architecture and compiler design; a large fraction of the reference manuals for most recent instruction set architectures is devoted to the memory model. Section 5.4 presents what are called "snooping" or "snoopy" protocols for bus-based machines and shows how they satisfy the conditions for coherence as well as for a useful consistency model. The basic design space of snooping protocols is laid out in Section 5.5 and the operation of commonly used protocols is described at the state transition level. The techniques used for quantitative evaluation several design tradeoffs at this level are illustrated using aspects of the methodology for workload driven evaluation of Chapter 4.

The latter portions of the chapter examine the implications these cache coherent shared memory architectures have for software that runs on them. Section 5.6 examines how the low-level synchronization operations make use of the available hardware primitives on cache coherent multiprocessors, and how the algorithms can be tailored to use the machine efficiently. Section 5.7 discusses the implications for parallel programming more generally, putting together our knowledge of parallel programs from Chapters 2 and 3 and of bus-based cache-coherent architecture

from this chapter. In particular, it discusses how temporal and spatial data locality may be exploited to reduce cache misses and traffic on the shared bus.

5.2 Cache Coherence

Think for a moment about your intuitive model of what a memory should do. It should provide a set of locations holding values, and when a location is read it should return the latest value written to that location. This is the fundamental property of the memory abstraction that we rely on in sequential programs when we use memory to communicate a value from a point in a program where it is computed to other points where it is used. We rely on the same property of a memory system when using a shared address space to communicate data between threads or processes running on one processor. A read returns the latest value written to the location, regardless of which process wrote it. Caching does not interfere with the use of multiple processes on one processor, because they all see the memory through the same cache hierarchy. We would also like to rely on the same property when the two processes run on different processors that share a memory. That is, we would like the results of a program that uses multiple processes to be no different when the processes run on different physical processors than when they run (interleaved or multiprogrammed) on the same physical processor. However, when two processes see the shared memory through different caches, there is a danger that one may see the new value in its cache while the other still sees the old value.

5.2.1 The Cache Coherence Problem

The cache coherence problem in multiprocessors is both pervasive and performance critical. The problem is illustrated by the following example.

Example 5-1   Figure 5-3 shows three processors with caches connected via a bus to shared main memory. A sequence of accesses to location u is made by the processors. First, processor P1 reads u from main memory, bringing a copy into its cache. Then processor P3 reads u from main memory, bringing a copy into its cache. Then processor P3 writes location u changing its value from 5 to 7. With a write-through cache, this will cause the main memory location to be updated; however, when processor P1 reads location u again (action 4), it will unfortunately read the stale value 5 from its own cache instead of the correct value 7 from main memory. What happens if the caches are writeback instead of write through?

Answer   The situation is even worse with writeback caches. P3's write would merely set the dirty (or modified) bit associated with the cache block holding location u and would not update main memory right away. Only when this cache block is subsequently replaced from P3's cache would its contents be written back to main memory. Not only will P1 read the stale value, but when processor P2 reads location u (action 5) it will miss in its cache and read the stale value of 5 from main memory, instead of 7. Finally, if multiple processors write distinct values to location u in their write-back caches, the final value that will reach main memory

will be determined by the order in which the cache blocks containing u are replaced, and will have nothing to do with the order in which the writes to u occur.
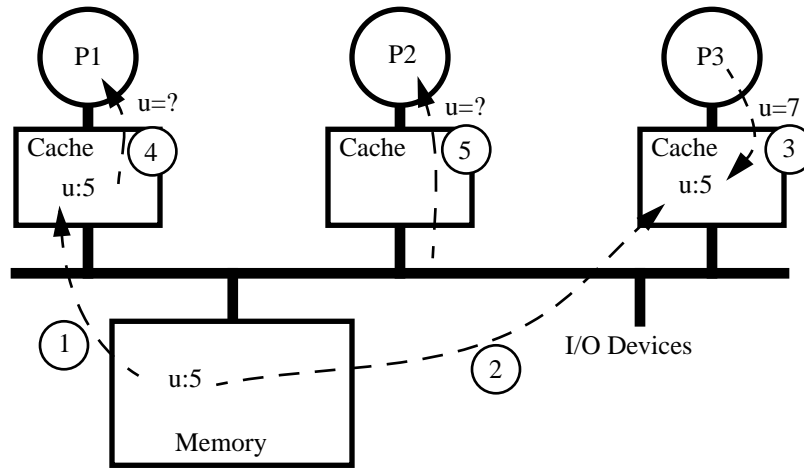
Figure 5-3 Example cache coherence problem

The figure shows three processors with caches connected by a bus to main memory. "u" is a location in memory whose contents are being read and written by the processors. The sequence in which reads and writes are done is indicated by the number listed inside the circles placed next to the arc. It is easy to see that unless special action is taken when P3 updates the value of u to 7, P1 will subsequently continue to read the stale value out of its cache, and P2 will also read a stale value out of main memory.

Cache coherence problems arise even in uniprocessors when I/O operations occur. Most I/O transfers are performed by DMA devices that move data between memory and the peripheral component. A DMA device sitting on the main memory bus may read a stale value for a location in main memory, because the latest value for that location is in the processor's write-back cache. Similarly, when the DMA device writes to a location in main memory, unless special action is taken, the processor may continue to see the old value if that location was previously present in its cache. Since I/O operations are much less frequent than memory operations, several simple solutions have been adopted in uniprocessors. For example, segments of memory space used for I/O may be marked as uncacheable, or the processor may always use uncached loads-stores for locations used to communicate with I/O devices. For I/O devices that transfer large blocks of data, such as disks, operating system support is often enlisted to ensure coherence. In many systems the page of memory from/to which the data is to be transferred is flushed by the operating system from the processor's cache before the I/O is allowed to proceed. In still other systems, all I/O traffic is made to flow through the processor cache hierarchy, thus maintaining coherence. This, of course, pollutes the cache hierarchy with data that may not be of immediate interest to the processor. Today, essentially all microprocessors provide support for cache coherence; in addition to making the chip "multiprocessor ready", this also solves the I/O coherence problem.

Clearly, the behavior described in Example 5-1 violates our intuitive notion of what a memory should do. Rreading and writing of shared variables is expected to be a frequent event in multiprocessors; it is the way that multiple processes belonging to a parallel application communicate with each other, so we do not want to disallow caching of shared data or invoke the operating system on all shared references. Rather, cache coherence needs to be addressed as a basic hardware design issue; for example, stale cached copies of a shared location must be eliminated when the location is modified, by either invalidating them or updating them with the new value. The oper-

ating system itself benefits greatly from transparent, hardware-supported coherent sharing of its data structures, and is in fact one of the most widely used parallel applications on these machines.

Before we explore techniques to provide coherence, it is useful to define the coherence property more precisely. Our intuitive notion, "each read should return the last value written to that location," is problematic for parallel architecture because "last" may not be well defined. Two different processors might write to the same location at the same instant, or one processor may write so soon after another that due to speed of light and other factors, there is not time to propagate the earlier update to the later writer. Even in the sequential case, "last" is not a chronological notion, but latest *in program order*. For now, we can think of program order in a single process as the order in which memory operations are presented to the processor by the compiler. The subtleties of program order will be elaborated on later, in Section 5.3. The challenge in the parallel case is that program order is defined for the operations in each individual process, and we need to make sense of the collection of program orders.

Let us first review the definitions of some terms in the context of uniprocessor memory systems, so we can extend the definitions for multiprocessors. By *memory operation*, we mean a single read (load), write (store), or read-modify-write access to a memory location. Instructions that perform multiple reads and writes, such as appear in many complex instruction sets, can be viewed as broken down into multiple memory operations, and the order in which these memory operations are executed is specified by the instruction. These memory operations within an instruction are assumed to execute *atomically* with respect to each other in the specified order, i.e. all aspects of one appear to execute before any aspect of the next. A memory operation *issues* when it leaves the processor's internal environment and is presented to the memory system, which includes the caches, write-buffers, bus, and memory modules. A very important point is that the processor only observes the state of the memory system by issuing memory operations; thus, our notion of what it means for a memory operation to be *performed* is that it appears to have taken place from the perspective of the processor. A write operation is said to *perform with respect to the processor* when a subsequent read by the processor returns the value produced by either that write or a later write. A read operation is said to perform with respect to the processor when subsequent writes issued by that processor cannot affect the value returned by the read. Notice that in neither case do we specify that the physical location in the memory chip has actually been accessed. Also, 'subsequent' is well defined in the sequential case, since reads and writes are ordered by the program order.

The same definitions for operations performing with respect to a processor apply in the parallel case; we can simply replace "the processor" by "a processor" in the definitions. The challenge for ordering, and for the intuitive notions of 'subsequent' and 'last', now is that we do not have one program order; rather, we have separate program orders for every process and these program orders interact when accessing the memory system. One way to sharpen our intuitive notion of a coherent memory system is to picture what would happen if there were no caches. Every write and every read to a memory location would access the physical location at main memory, and would be performed with respect to all processors at this point, so the memory would impose a serial order on all the read and write operations to the location. Moreover, the reads and writes to the location from any individual processor should be in program order within this overall serial order. We have no reason to believe that the memory system should interleave independent accesses from different processors in a particular way, so any interleaving that preserves the individual program orders is reasonable. We do assume some basic fairness; eventually the operations from each processor should be performed. Thus, our intuitive notion of "last" can be viewed as most recent in some hypothetical serial order that maintains the properties discussed above.

Since this serial order must be consistent, it is important that all processors see the writes to a location in the same order (if they bother to look, i.e. to read the location).

Of course, the total order need not actually be constructed at any given point in the machine while executing the program. Particularly in a system with caches, we do not want main memory to see all the memory operations, and we want to avoid serialization whenever possible. We just need to make sure that the program behaves as if some serial order was enforced.

More formally, we say that a multiprocessor memory system is *coherent* if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processors into a total order) which is consistent with the results of the execution and in which:

1. operations issued by any particular processor occur in the above sequence in the order in which they were issued to the memory system by that processor, and

2. the value returned by each read operation is the value written by the last write to that location in the above sequence.

Implicit in the definition of coherence is the property that all writes to a location (from the same or different processes) are seen in the same order by all processes. This property is called *write serialization*. It means that if read operations by processor P1 to a location see the value produced by write *w1* (from P2, say) before the value produced by write *w2* (from P3, say), then reads by another processor P4 (or P2 or P3) should also not be able to see *w2* before *w1*. There is no need for an analogous concept of read serialization, since the effects of reads are not visible to any processor but the one issuing the read.

The results of a program can be viewed as the values returned by the read operations in it, perhaps augmented with an implicit set of reads to all locations at the end of the program. From the results, we cannot determine the order in which operations were actually executed by the machine, but only orders in which they appear to execute. In fact, it is not even important in what order things actually happen in the machine or when which bits change, since this is not detectable; all that matters is the order in which things appear to happen, as detectable from the results of an execution. This concept will become even more important when we discuss memory consistency models. Finally, the one additional definition that we will need in the multiprocessor case is that of an operation completing: A read or write operation is said to *complete* when it has performed with respect to all processors.

5.2.2 Cache Coherence Through Bus Snooping

A simple and elegant solution to the cache coherence problem arises from the very nature of a bus. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction, for example every read or write on the shared bus. When a processor issues a request to its cache, the controller examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having each cache controller 'snoop' on the bus and monitor the transactions, as illustrated in Figure 5-4 [Goo83]. The snooping cache controller also takes action if a bus transaction is relevant to it, i.e. involves a memory block of which it has a copy in its cache. In fact, since the allocation and replacement of data in caches are managed at the granularity of a cache block (usually several words long) and cache misses fetch a block of data, most often coherence is maintained at

the granularity of a cache block as well. That is, either an entire cache block is in valid state or none of it is. Thus, a cache block is the granularity of allocation in the cache, of data transfer between caches and of coherence.

The key properties of a bus that support coherence are the following: All transactions that appear on the bus are visible to all cache controllers, and they are visible to the controllers in the same order (the order in which they appear on the bus). A coherence protocol must simply guarantee that all the necessary transactions in fact appear on the bus, in response to memory operations, and that the controllers take the appropriate actions when they see a relevant transaction.

The simplest illustration of maintaining coherence is a system that has single-level write-through caches. It is basically the approach followed by the first commercial bus-based SMPs in the mid 80's. In this case, every write operation causes a write transaction to appear on the bus, so every cache observes every write. If a snooping cache has a copy of the block, it either invalidates or updates its copy. Protocols that invalidate other cached copies on a write are called *invalidation-based protocols*, while those that update other cached copies are called *update-based protocols*. In either case, the next time the processor with the invalidated or updated copy accesses the block, it will see the most recent value, either through a miss or because the updated value is in its cache. The memory always has valid data, so the cache need not take any action when it observes a read on the bus.

Example 5-2   Consider the scenario that was shown in Figure 5-3. Assuming write through caches, show how the bus may be used to provide coherence using an invalidation-based protocol.

Answer   When processor P3 writes 7 to location u, P3's cache controller generates a bus transaction to update memory. Observing this bus transaction as relevant, P1's cache controller invalidates its own copy of block containing u. The main memory controller will update the value it has stored for location u to 7. Subsequent reads to u from processors P1 and P2 (actions 4 and 5) will both miss in their private caches and get the correct value of 7 from the main memory.

In general, a snooping-based cache coherence scheme ensures that:

- all "necessary" transactions appear on the bus, and
- each cache monitors the bus for relevant transactions and takes suitable actions.

The check to determine if a bus transaction is relevant to a cache is essentially the same tag match that is performed for a request from the processor. The suitable action may involve invalidating or updating the contents or state of that memory block, and/or supplying the latest value for that memory block from the cache to the bus.

A *snoopy cache coherence protocol* ties together two basic facets of computer architecture: bus transactions and the state transition diagram associated with a cache block. Recall, a bus transaction consists of three phases: arbitration, command/address and data. In the arbitration phase, devices that desire to perform (or master) a transaction assert their bus request and the bus arbiter selects one of these and responds by asserting its grant signal. Upon grant, the device places the command, e.g. read or write, and the associated address on the bus command and address lines.
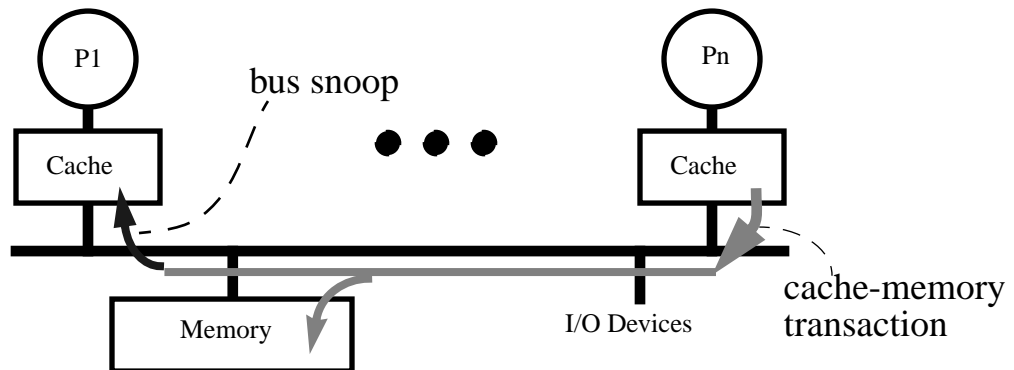
Figure 5-4 Snoopy cache-coherent multiprocessor

Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously "snoops" on the bus watching for relevant transactions, and updates its state suitably to keep its local cache coherent.

All devices observe the address and one of them recognizes that it is responsible for the particular address. For a read transaction, the address phase is followed by data transfer. Write transactions vary from bus to bus in whether the data is transferred during or after the address phase. For most busses, the slave device can assert a wait signal to hold off the data transfer until it is ready. This wait signal is different from the other bus signals, because it is a wired-OR across all the processors, i.e., it is a logical 1 if any device asserts it. The master does not need to know which slave device is participating in the transfer, only that there is one and it is not yet ready.

The second basic notion is that each block in a uniprocessor cache has a state associated with it, along with the tag and data, indicating the disposition of the block, e.g., invalid, valid, dirty. The cache policy is defined by the *cache block state transition diagram*, which is a finite state machine. Transitions for a block occur upon access to an address that maps to the block. For a write-through, write-no-allocate cache [HeP90] only two states are required: valid and invalid. Initially all the blocks are invalid. (Logically, all memory blocks that are not resident in the cache can be viewed as being in either a special "not present" state, or in "invalid" state.) When a processor read operation misses, a bus transaction is generated to load the block from memory and the block is marked valid. Writes generate a bus transaction to update memory and the cache block if it is present in valid state. Writes never change the state of the block. (If a block is replaced, it may be marked invalid until the memory provides the new block, whereupon it becomes valid.) A write-back cache requires an additional state, indicating a "dirty" or modified block.

In a multiprocessor system, a block has a state in each cache, and these cache states change according to the state transition diagram. Thus, we can think of a block's "cache state" as being a vector of *p* states instead of a single state, where *p* is the number of caches. The cache state is manipulated by a set of *p* distributed finite state machines, implemented by the cache controllers. The state machine or state transition diagram that governs the state changes is the same for all blocks and all caches, but the current states of a block in different caches is different. If a block is not present in a cache, we can assume it to be in a special "not present" state or even in invalid state.

In a snoopy cache coherence scheme, each cache controller receives two sets of inputs: the processor issues memory requests, and the bus snooper informs about transactions from other caches. In response to either, the controller updates the state of the appropriate block in the cache according to the current state and the state transition diagram. It responds to the processor with requested data, potentially generating new bus transactions to obtain the data. It responds to bus transactions generated by others by updating its state, and sometimes intervenes in completing the transaction. Thus, a snoopy protocol is a distributed algorithm represented by a collection of such cooperating finite state machines. It is specified by the following components:

1. the set of states associated with memory blocks in the local caches;

2. the state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction, and produces as output the next state for the cache block; and

3. the actual actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, cache, and processor design.

The different state machines for a block do not operate independently, but are coordinated by bus transactions.

A simple invalidation-based protocol for a coherent write-through no-allocate cache is described by the state diagram in Figure 5-5. As in the uniprocessor case, each cache block has only two states: invalid (I) and valid (V) (the "not present" state is assumed to be the same as invalid). The transitions are marked with the input that causes the transition and the output that is generated with the transition. For example, when a processor read misses in the cache a BusRd transaction is generated, and upon completion of this transaction the block transitions up to the "valid" state. Whenever the processor issues a write to a location, a bus transaction is generated that updates that location in main memory with no change of state. The key enhancement to the uniprocessor state diagram is that when the bus snooper sees a write transaction for a memory block that is cached locally, it sets the cache state for that block to invalid thereby effectively discarding its copy. Figure 5-5 shows this bus-induced transition with a dashed arc. By extension, if any one cache generates a write for a block that is cached by any of the others, all of the others will invalidate their copies. The collection of caches thus implements a single writer, multiple reader discipline.

To see that this simple write-through invalidation protocol provides coherence, we need to show that a total order on the memory operations for a location can be constructed that satisfies the two conditions. Assume for the present discussion that memory operations are *atomic*, in that only one transaction is in progress on the bus at a time and a processor waits until its transaction is finished before issuing another memory operation. That is, once a request is placed on the bus, all phases of the transaction including the data response complete before any other request from any processor is allowed access to the bus. With single-level caches, it is natural to assume that invalidations are applied to the caches, and the write completes, during the bus transaction itself. (These assumptions will be relaxed when we look at protocol implementations in more detail and as we study high performance designs with greater concurrency.) We may assume that the memory handles writes and reads in the order they are presented to it by the bus. In the write-through protocol, all writes go on the bus and only one bus transaction is in progress at a time, so all writes to a location are serialized (consistently) by the order in which they appear on the shared bus, called the *bus order*. Since each snooping cache controller performs the invalidation during the bus transaction, invalidations are performed by all cache controllers in bus order.
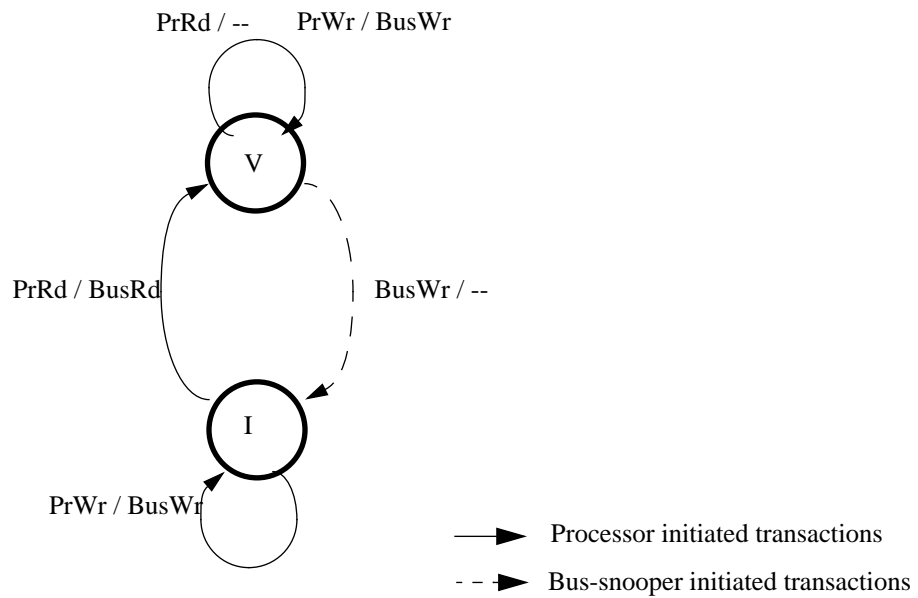
Figure 5-5 Snoopy coherence for a multiprocessor with write-through no-write-allocate caches.

There are two states, valid (V) and invalid (I) with intuitive semantics. The notation A/B (e.g. PrRd/BusRd) means if you observe A then generate transaction B. From the processor side, the requests can be read (PrRd) or write (PrWr). From the bus side, the cache controller may observe/generate transactions bus read (BusRd) or bus write (BusWr).

Processors "see" writes through read operations. However, reads to a location are not completely serialized, since read hits may be performed independently and concurrently in their caches without generating bus transactions. To see how reads may be inserted in the serial order of writes, and guarantee that all processors see writes in the same order (write serialization), consider the following scenario. A read that goes on the bus (a read miss) is serialized by the bus along with the writes; it will therefore obtain the value written by the most recent write to the location in bus order. The only memory operations that do not go on the bus are read hits. In this case, the value read was placed in the cache by either the most recent write to that location by the same processor, or by its most recent read miss (in program order). Since both these sources of the value appear on the bus, read hits also see the values produced in the consistent bus order.

More generally, we can easily construct a hypothetical serial order by observing the following partial order imposed by the protocol:

A memory operation M_2 is subsequent to a memory operation M_1 if the operations are issued by the same processor and M_2 follows M_1 in program order.

A read operation is subsequent to a write operation W if the read generates a bus transaction that follows that for W.

A write operation is subsequent to a read or write operation M if M generates a bus transaction and the transaction for the write follows that for M.

A write operation is subsequent to a read operation if the read does not generate a bus transaction (is a hit) and is not already separated from the write by another bus transaction.

The "subsequent" ordering relationship is transitive. An illustration of this partial order is depicted in Figure 5-6, where the bus transactions associated with writes segment the individual program orders. The partial order does not constrain the ordering of read bus transactions from different processors that occur between two write transactions, though the bus will likely establish a particular order. In fact, any interleaving of read operations in the segment between two writes is a valid serial order, as long as it obeys program order.
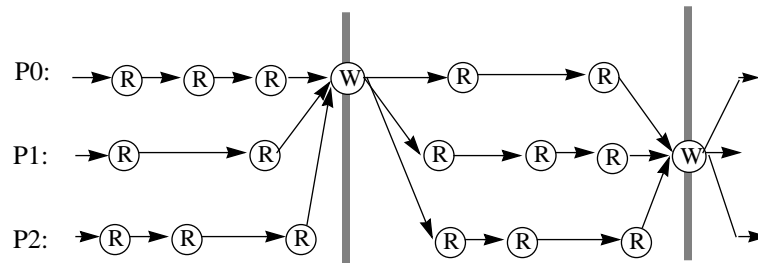
Figure 5-6 Partial order of memory operations for an execution with the write-through invalidation protocol

Bus transactions define a global sequence of events, between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving the processor orders within each segment.

Of course, the problem with this simple write-through approach is that every store instruction goes to memory, which is why most modern microprocessors use write-back caches (at least at the level closest to the bus). This problem is exacerbated in the multiprocessor setting, since every store from every processor consumes precious bandwidth on the shared bus, resulting in poor scalability of such multiprocessors as illustrated by the example below.

Example 5-3   Consider a superscalar RISC processor issuing two instructions per cycle running at 200MHz. Suppose the average CPI (clocks per instruction) for this processor is 1, 15% of all instructions are stores, and each store writes 8 bytes of data. How many processors will a GB/s bus be able to support without becoming saturated?

Answer   A single processor will generate 30 million stores per second (0.15 stores per instruction * 1 instruction per cycle * 1,000,000/200 cycles per second), so the total write-through bandwidth is 240Mbytes of data per second per processor (ignoring address and other information, and ignoring read misses). A GB/s bus will therefore support only about four processors.

For most applications, a write-back cache would absorb the vast majority of the writes. However, if writes do not go to memory they do not generate bus transactions, and it is no longer clear how the other caches will observe these modifications and maintain cache coherence. A somewhat more subtle concern is that when writes are allowed to occur into different caches concurrently there is no obvious order to the sequence of writes. We will need somewhat more sophisticated cache coherence protocols to make the "critical" events visible to the other caches and ensure write serialization.

Before we examine protocols for write-back caches, let us step back to the more general ordering issue alluded to in the introduction to this chapter and examine the semantics of a shared address space as determined by the memory consistency model.

5.3 Memory Consistency

Coherence is essential if information is to be transferred between processors by one writing a location that the other reads. Eventually, the value written will become visible to the reader, indeed all readers. However, it says nothing about when the write will become visible. Often, in writing a parallel program we want to ensure that a read returns the value of a particular write, that is we want to establish an order between a write and a read. Typically, we use some form of event synchronization to convey this dependence and we use more than one location.

Consider, for example, the code fragments executed by processors P1 and P2 in Figure 5-7, which we saw when discussing point-to-point event synchronization in a shared address space in Chapter 2. It is clear that the programmer intends for process P2 to spin idly until the value of the shared variable `flag` changes to 1, and to then print the value of variable A as 1, since the value of A was updated before that of `flag` by process P1. In this case, we use accesses to another location (`flag`) to preserve order among different processes' accesses to the same location (A). In particular, we assume that the write of A becomes visible to P2 before the write to `flag`, and that the read of flag by P2 that breaks it out of its while loop completes before its read of A. These program orders within P1 and P2's accesses are not implied by coherence, which, for example, only requires that the new value for A eventually become visible to processor P2, not necessarily before the new value of `flag` is observed.

```
P1                      P2
/* Assume initial value of A and flag is 0 */
A = 1;                  while (flag == 0); /* spin idly */
flag = 1;               print A;
```

Figure 5-7 Requirements of event synchronization through flags

The figure shows two processors concurrently executing two distinct code fragments. For programmer intuition to be maintained, it must be the case that the printed value of A is 1. The intuition is that because of program order, if flag equals 1 is visible to processor P2, then it must also be the case that A equals 1 is visible to P2.

The programmer might try to avoid this issue by using a barrier, as shown in Figure 5-8. We expect the value of A to be printed as 1, since A was set to 1 before the barrier (note that a print statement is essentially a read). There are two potential problems even with this approach. First, we are adding assumptions to the meaning of the barrier: Not only do processes wait at the barrier till all have arrived, but until all writes issued prior to the barrier have become visible to other processors. Second, a barrier is often built using reads and writes to ordinary shared variables (e.g. `b1` in the figure) rather than with specialized hardware support. In this case, as far as the machine is concerned it sees only accesses to different shared variables; coherence does not say anything at all about orders among these accesses.

```
     P1                                  P2
     /* Assume initial value of A is 0 */
     A = 1;                   ...
- — - BARRIER(b1)- — — — — — — -BARRIER(b1) - — -
                                      print A;
```

Figure 5-8 Maintaining orders among accesses to a location using explicit synchronization through barriers.

Clearly, we expect more from a memory system than "return the last value written" for each location. To establish order among accesses to the location (say A) by different processes, we sometimes expect a memory system to respect the order of reads and writes to *different* locations (A and flag or A and b1) issued by a given process. Coherence says nothing about the order in which the writes issued by P1 become visible to P2, since these operations are to different locations. Similarly, it says nothing about the order in which the reads issued to different locations by P2 are performed relative to P1. Thus, coherence does not in itself prevent an answer of 0 being printed by either example, which is certainly not what the programmer had in mind.

In other situations, the intention of the programmer may not be so clear. Consider the example in Figure 5-9. The accesses made by process P1 are ordinary writes, and A and B are not used as

```
     P1                     P2
     /* Assume initial values of A and B are 0 */
     (1a) A = 1;            (2a) print B;
     (1b) B = 2;            (2b) print A;
```

Figure 5-9 Orders among accesses without synchronization.

synchronization variables. We may intuitively expect that if the value printed for B is 2 then the value printed for A is 1 as well. However, the two print statements read different locations before printing them, so coherence says nothing about how the writes by P1 become visible. (This example is a simplification of Dekker's algorithm to determine which of two processes arrives at a critical point first, and hence ensure mutual exclusion. It relies entirely on writes to distinct locations becoming visible to other processes in the order issued.) Clearly we need something more than coherence to give a shared address space a clear semantics, i.e., an ordering model that programmers can use to reason about the possible results and hence correctness of their programs.

A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e. to become visible to the processors) with respect to one another. This includes operations to the same locations or to different locations, and by the same process or different processes, so memory consistency subsumes coherence.

5.3.1 Sequential Consistency

In discussing the fundamental design issues for a communication architecture in Chapter 1 (Section 1.4), we described informally a desirable ordering model for a shared address space: the reasoning one goes through to ensure a multithreaded program works under any possible interleaving on a uniprocessor should hold when some of the threads run in parallel on different processors. The ordering of data accesses within a process was therefore the program order, and that across processes was some interleaving of the program orders. That is, the multiprocessor case should not be able to cause values to become visible in the shared address space that no interleaving of accesses from different processes can generate. This intuitive model was formalized by Lamport as *sequential consistency* (SC), which is defined as follows [Lam79].[1]

> *Sequential Consistency* A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Figure 5-10 depicts the abstraction of memory provided to programmers by a sequentially consistent system [AdG96]. Multiple processes *appear* to share a *single* logical memory, even though in the real machine main memory may be distributed across multiple processors, each with their private caches and write buffers. Every processor appears to issue and complete memory operations one at a time and atomically in program order—that is, a memory operation does not appear to be issued until the previous one has completed—and the common memory appears to service these requests one at a time in an interleaved manner according to an arbitrary (but hopefully fair) schedule. Memory operations appear *atomic* in this interleaved order; that is, it
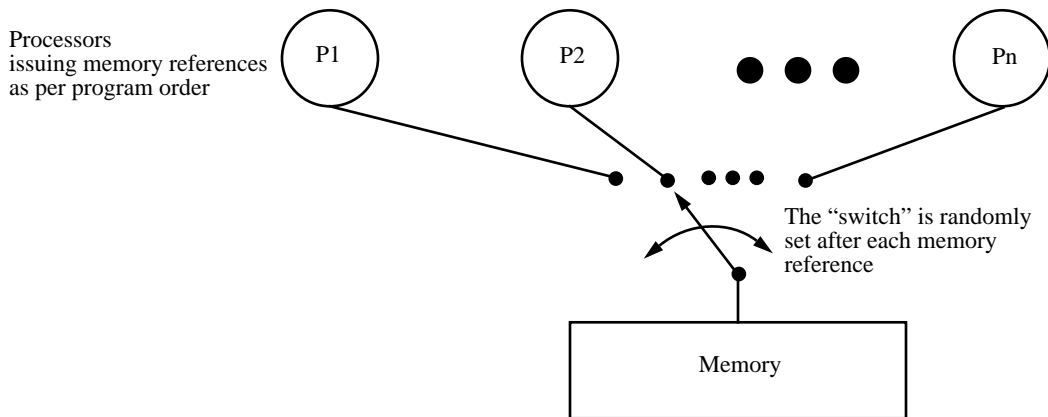
Figure 5-10 Programmer's abstraction of the memory subsystem under the sequential consistency model.

The model completely hides the underlying concurrency in the memory-system hardware, for example, the possible existence of distributed main memory, the presence of caches and write buffers, from the programmer.

---

1. Two closely related concepts in the software context are serializability[Papa79] for conccurent updates to a database and linearizability[HeWi87] for concurrent objects.

should appear globally (to all processors) as if one operation in the consistent interleaved order executes and completes before the next one begins.

As with coherence, it is not important in what order memory operations actually issue or even complete. What matters is that they appear to complete in an order that does not violate sequential consistency. In the example in Figure 5-9, under SC the result (0,2) for (A,B) would not be allowed under sequential consistency—preserving our intuition—since it would then appear that the writes of A and B by process P1 executed out of program order. However, the memory operations may actually execute and complete in the order 1b, 1a, 2b, 2a. It does not matter that they actually complete out of program order, since the results of the execution (1,2) is the same as if the operations were executed and completed in program order. On the other hand, the actual execution order 1b, 2a, 2b, 1a would not be sequentially consistent, since it would produce the result (0,2) which is not allowed under SC. Other examples illustrating the intuitiveness of sequential consistency can be found in Exercise 5.4. Note that sequential consistency does not obviate the need for synchronization. SC allows operations from different processes to be interleaved arbitrarily and at the granularity of individual instructions. Synchronization is needed if we want to preserve atomicity (mutual exclusion) across multiple memory operations from a process, or if we want to enforce certain orders in the interleaving across processes.

The term "program order" also bears some elaboration. Intuitively, program order for a process is simply the order in which statements appear in the source program; more specifically, the order in which memory operations appear in the assembly code listing that results from a straightforward translation of source statements one by one to assembly language instructions. This is not necessarily the order in which an optimizing compiler presents memory operations to the hardware, since the compiler may reorder memory operations (within certain constraints such as dependences to the same location). The programmer has in mind the order of statements in the program, but the processor sees only the order of the machine instructions. In fact, there is a "program order" at each of the interfaces in the communication architecture—particularly the programming model interface seen by the programmer and the hardware-software interface—and ordering models may be defined at each. Since the programmer reasons with the source program, it makes sense to use this to define program order when discussing memory consistency models; that is, we will be concerned with the consistency model presented by the system to the programmer.

Implementing SC requires that the system (software and hardware) preserve the intuitive constraints defined above. There are really two constraints. The first is the *program order* requirement discussed above, which means that it must appear as if the memory operations of a process become visible—to itself and others—in program order. The second requirement, needed to guarantee that the total order or interleaving is consistent for all processes, is that the operations appear atomic; that is, it appear that one is completed with respect to all processes before the next one in the total order is issued. The tricky part of this second requirement is making writes appear atomic, especially in a system with multiple copies of a block that need to be informed on a write. *Write atomicity*, included in the definition of SC above, implies that the position in the total order at which a write appears to perform should be the same with respect to all processors. It ensures that nothing a processor does *after* it has seen the new value produced by a write becomes visible to other processes before they too have seen the new value for that write. In effect, while coherence (write serialization) says that writes to the same location should appear to all processors to have occurred in the same order, sequential consistency says that all writes (to any location) should appear to all processors to have occurred in the same order. The following example shows why write atomicity is important.

Example 5-4   Consider the three processes in Figure 5-11. Show how not preserving write atomicity violates sequential consistency.

Answer   Since P2 waits until A becomes 1 and then sets B to 1, and since P3 waits until B becomes 1 and only then reads value of A, from transitivity we would infer that P3 should find the value of A to be 1. If P2 is allowed to go on past the read of A and write B before it is guaranteed that P3 has seen the new value of A, then P3 may read the new value of B but the old value of A from its cache, violating our sequentially consistent intuition.

P1   P2   P3

A=1; ─────────▶ while (A==0);
                    B=1; ─────────────────▶ while (B==0);
                                            print A;
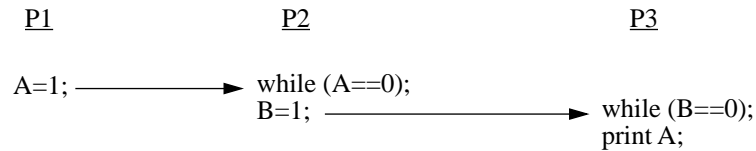
Figure 5-11 Example illustrating the importance of write atomicity for sequential consistency.

Each process's program order imposes a partial order on the set of all operations; that is, it imposes an ordering on the subset of the operations that are issued by that process. An interleaving (in the above sense) of the operations from different processes defines a total order on the set of all operations. Since the exact interleaving is not defined by SC, interleaving the partial (program) orders for different processes may yield a large number of possible total orders. The following definitions apply:

Sequentially Consistent Execution An execution of a program is said to be sequentially consistent if the results it produces are the same as those produced by any one of these possible total orders (interleavings as defined earlier). That is, there should exist a total order or interleaving of program orders from processes that yields the same result as that actual execution.

Sequentially Consistent System A system is sequentially consistent if any possible execution on that system corresponds to (produces the same results as) some possible total order as defined above.

Of course, an implicit assumption throughout is that a read returns the last value that was written to that same location (by any process) in the interleaved total order.

5.3.2 Sufficient Conditions for Preserving Sequential Consistency

It is possible to define a set of sufficient conditions that the system should obey that will guarantee sequential consistency in a multiprocessor—whether bus-based or distributed, cache-coherent or not. The following set, adapted from their original form [DSB86,ScD87], are commonly used because they are relatively simple without being overly restrictive:

1. Every process issues memory requests in the order specified by the program.

2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.

3. After a read operation is issued, the issuing process waits for the read to complete, *and* for the write whose value is being returned by the read to complete, before issuing its next operation. That is, if the write whose value is being returned has performed with respect to this processor (as it must have if its value is being returned) then the processor should wait until the write has performed with respect to all processors.

The third condition is what ensures write atomicity, and it is quite demanding. It is not a simple local constraint, because the load must wait until the logically preceding store has become globally visible. Note that these are sufficient, rather than necessary conditions. Sequential consistency can be preserved with less serialization in many situations. This chapter uses these conditions, but exploits other ordering constraints in the bus-based design to establish completion early.

With program order defined in terms of the source program, it is clear that for these conditions to be met the compiler should not change the order of memory operations that it presents to the processor. Unfortunately, many of the optimizations that are commonly employed in both compilers and processors violate the above sufficient conditions. For example, compilers routinely reorder accesses to different locations within a process, so a processor may in fact issue accesses out of the program order seen by the programmer. Explicitly parallel programs use uniprocessor compilers, which are concerned only about preserving dependences to the same location. Advanced compiler optimizations designed to improve performance—such as common sub-expression elimination, constant propagation, and loop transformations such as loop splitting, loop reversal, and blocking [Wol96]—can change the order in which different locations are accessed or even eliminate memory references.[1] In practice, to constrain compiler optimizations multithreaded and parallel programs annotate variables or memory references that are used to preserve orders. A particularly stringent example is the use of the `volatile` qualifier in a variable declaration, which prevents the variable from being register allocated or any memory operation on the variable from being reordered with respect to operations before or after it.

Example 5-5   How would reordering the memory operations in Figure 5-7 affect semantics in a sequential program (only one of the processes running), in a parallel program running on a multiprocessor, and in a threaded program in which the two processes are interleaved on the same processor. How would you solve the problem?

Answer   The compiler may reorder the writes to A and `flag` with no impact on a sequential program. However, this can violate our intuition for both parallel programs and concurrent uniprocessor programs. In the latter case, a context switch can happen between the two reordered writes, so the process switched in may see the update to `flag` without seeing the update to A. Similar violations of intuition occur if the compiler reorders the reads of flag and A. For many compilers, we can avoid these reorderings by declaring the variable `flag` to be of type `volatile integer`

---

1. Note that register allocation, performed by modern compilers to eliminate memory operations, can be dangerous too. In fact, it can affect coherence itself, not just memory consistency. For the flag synchronization example in Figure 5-7, if the compiler were to register allocate the flag variable for process P2, the process could end up spinning forever: The cache-coherence hardware updates or invalidates only the memory and the caches, not the registers of the machine, so the write propagation property of coherence is violated.

instead of just `integer`. Other solutions are also possible, and will be discussed in Chapter 9.

Even if the compiler preserves program order, modern processors use sophisticated mechanisms like write buffers, interleaved memory, pipelining and out-of-order execution techniques [HeP90]. These allow memory operations from a process to issue, execute and/or complete out of program order. These architectural and compiler optimizations work for sequential programs because there the appearance of program order requires that dependences be preserved only among accesses to the same memory location, as shown in Figure 5-12.
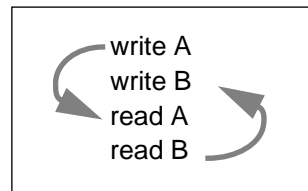
Figure 5-12 Preserving orders in a sequential program running on a uniprocessor.

Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

Preserving sequential consistency in multiprocessors is quite a strong requirement; it limits compiler reordering and out of order processing techniques. The problem is that the out of order processing of operations to shared variables by a process can be detected by other processes. Several weaker consistency models have been proposed, and we will examine these in the context of large scale shared address space machines in Chapter 6. For the purposes of this chapter, we will assume the compiler does not perform optimizations that violate the sufficient conditions for sequential consistency, so the program order that the processor sees is the same as that seen by the programmer. On the hardware side, to satisfy the sufficient conditions we need mechanisms for a processor to detect completion of its writes so it may proceed past them—completion of reads is easy, it is when the data returns to the processor—and mechanisms to preserve write atomicity. For all the protocols and systems considered in this chapter, we will see how they satisfy coherence (including write serialization), how they can satisfy sequential consistency (in particular, how write completion is detected and write atomicity is guaranteed), and what shortcuts can be taken while still satisfying the sufficient conditions.

The serialization imposed by transactions appearing on the shared bus is very useful in ordering memory operations. The reader should verify that the 2-state write-through invalidate protocol discussed above actually provides sequential consistency, not just coherence, quite easily. The key observation is that writes and read misses to all locations, not just to individual locations, are serialized in bus order. When a read obtains the value of a write, the write is guaranteed to have completed, since it caused a previous bus transaction. When a write is performed, all previous writes have completed.

5.4 Design Space for Snooping Protocols

The beauty of snooping-based cache coherence is that the entire machinery for solving a difficult problem boils down to a small amount of extra interpretation on events that naturally occur in the system. The processor is completely unchanged. There are no explicit coherence operations inserted in the program. By extending the requirements on the cache controller and exploiting the properties of the bus, the loads and stores that are inherent to the program are used implicitly to keep the caches coherent and the serialization of the bus maintains consistency. Each cache controller observes and interprets the memory transactions of others to maintain its internal state. Our initial design point with write-through caches is not very efficient, but we are now ready to study the design space for snooping protocols that make efficient use of the limited bandwidth of the shared bus. All of these use write-back caches, allowing several processors to write to different blocks in their local caches concurrently without any bus transactions. Thus, extra care is required to ensure that enough information is transmitted over the bus to maintain coherence. We will also see how the protocols provide sufficient ordering constraints to maintain write serialization and a sequentially consistent memory model.

Recall that with a write-back cache on a uniprocessor, a processor write miss causes the cache to read the entire block from memory, update a word, and retain the block as *modified* (or *dirty*) so it may be written back on replacement. In a multiprocessor, this modified state is also used by the protocols to indicate exclusive ownership of the block by a cache. In general, a cache is said to be the *owner* of a block if it must supply the data upon a request for that block [SwS86]. A cache is said to have an *exclusive* copy of a block if it is the only cache with a valid copy of the block (main memory may or may not have a valid copy). Exclusivity implies that the cache may modify the block without notifying anyone else. If a cache does not have exclusivity, then it cannot write a new value into the block before first putting a transaction on the bus to communicate with others. The data may be in the writer's cache in a valid state, but since a transaction must be generated this is called a write miss just like a write to a block that is not present or invalid in the cache. If a cache has the block in modified state, then clearly it is both the owner and has exclusivity. (The need to distinguish ownership from exclusivity will become clear soon.)

On a write miss, then, a special form of transaction called a *read-exclusive* is used to tell other caches about the impending write and to acquire a copy of the block with exclusive ownership. This places the block in the cache in modified state, where it may now be written. Multiple processors cannot write the same block concurrently, which would lead to inconsistent values: The read-exclusive bus transactions generated by their writes will be serialized by the bus, so only one of them can have exclusive ownership of the block at a time. The cache coherence actions are driven by these two types of transactions: read and read-exclusive. Eventually, when a modified block is replaced from the cache, the data is written back to memory, but this event is not caused by a memory operation to that block and is almost incidental to the protocol. A block that is not in modified state need not be written back upon replacement and can simply be dropped, since memory has the latest copy. Many protocols have been devised for write-back caches, and we will examine the basic alternatives.

We also consider update-based protocols. Recall that in update-based protocols, when a shared location is written to by a processor its value is updated in the caches of all other processors holding that memory block[1]. Thus, when these processors subsequently access that block, they can do so from their caches with low latency. The caches of all other processors are updated with a single bus transaction, thus conserving bandwidth when there are multiple sharers. In contrast, with

invalidation-based protocols, on a write operation the cache state of that memory block in all other processors' caches is set to invalid, so those processors will have to obtain the block through a miss and a bus transaction on their next read. However, subsequent writes to that block by the same processor do not create any further traffic on the bus, until the block is read by another processor. This is attractive when a single processor performs multiple writes to the same memory block before other processors read the contents of that memory block. The detailed tradeoffs are quite complex and they depend on the workload offered to the machine; they will be illustrated quantitatively in Section 5.5. In general, invalidation-based strategies have been found to be more robust and are therefore provided as the default protocol by most vendors. Some vendors provide an update protocol as an option to be used selectively for blocks corresponding to specific data structures or pages.

The choices made for the protocol, update versus invalidate, and caching strategies directly affect the choice of states, the state transition diagram, and the associated actions. There is substantial flexibility available to the computer architect in the design task at this level. Instead of listing all possible choices, let us consider three common coherence protocols that will illustrate the design options.

5.4.1 A 3-state (MSI) Write-back Invalidation Protocol

The first protocol we consider is a basic invalidation-based protocol for write-back caches. It is very similar to the protocol that was used in the Silicon Graphics 4D series multiprocessor machines [BJS88]. The protocol uses the three states required for any write-back cache in order to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty). Specifically, the states are *invalid* (I), *shared* (S), and *modified* (M). Invalid has the obvious meaning. Shared means the block is present in unmodified state in this cache, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified, also called *dirty*, was discussed earlier; it means that only this processor has a valid copy of the block in its cache, the copy in main memory is stale, and no other cache may have a valid copy of the block (in either shared or modified state). Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated via a read-exclusive bus transaction. This transaction serves to order the write as well as cause the invalidations and hence ensure that the write becomes visible to others.

The processor issues two types of requests: reads (PrRd) and writes (PrWr). The read or write could be to a memory block that exists in the cache or to one that does not. In this latter case, the block currently in the cache will have to be replaced by the newly requested block, and if the existing block is in modified state its contents will have to be written back to main memory.

We assume that the bus allows the following transactions:

Bus Read (BusRd): The cache controller puts the address on the bus and asks for a copy that it does not intend to modify. The memory system (possibly another cache) supplies the data.

---

1. This is a write-broadcast scenario. Read-broadcast designs have also been investigated, in which the cache containing the modified copy flushes it to the bus on a read, at which point all other copies are updated too.

This transaction is generated by a PrRd that misses in the cache, and the processor expects a data response as a result.

Bus Read-Exclusive (BusRdX): The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. The memory system (possibly another cache) supplies the data. All other caches need to be invalidated. This transaction is generated by a PrWr to a block that is either not in the cache or is in the cache but not in modified state. Once the cache obtains the exclusive copy, the write can be performed in the cache. The processor may require an acknowledgment as a result of this transaction.

Writeback (BusWB): The cache controller puts the address and the contents for the memory block on the bus. The main memory is updated with the latest contents. This transaction is generated by the cache controller on a writeback; the processor does not know about it, and does not expect a response.

The Bus Read-Exclusive (sometimes called *read-to-own*) is a new transaction that would not exist except for cache coherence. The other new concept needed to support write-back protocols is that in addition to changing the state of cached blocks, the cache controller can intervene in the bus transaction and "flush" the contents of the referenced block onto the bus, rather than allow the memory to supply the data. Of course, the cache controller can also initiate new bus transactions as listed above, supply data for writebacks, or pick up data supplied by the memory system.
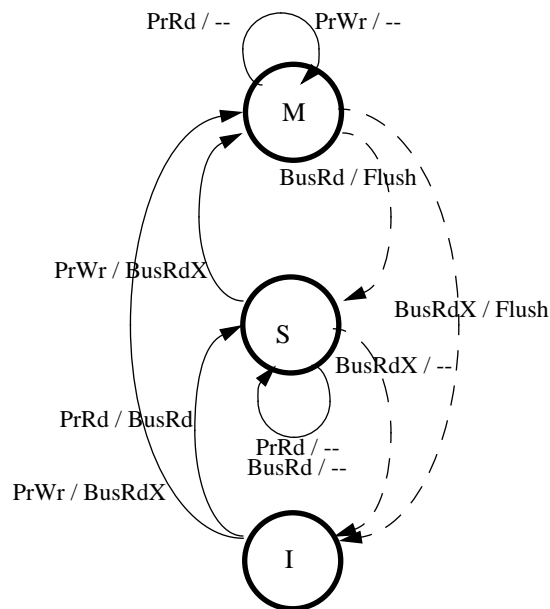
Figure 5-13 Basic 3-state invalidation protocol.

I, S, and M stand for Invalid, Shared, and Modified states respectively. The notation A / B means that if we observe from processor-side or bus-side event A, then in addition to state change, generate bus transaction or action B. "--" means null action. Transitions due to observed bus transactions are shown in dashed arcs, while those due to local processor actions are shown in bold arcs. If multiple A / B pairs are associated with an arc, it simply means that multiple inputs can cause the same state transition. For completeness, we need to specify actions from each state corresponding to each observable event. If such transitions are not shown, it means that they are uninteresting and no action needs to be taken. Replacements and the writebacks they may cause are not shown in the diagram for simplicity.

State Transitions

The state transition diagram that governs a block in each cache in this snoopy protocol is as shown in Figure 5-13. The states are organized so that the closer the state is to the top the more tightly the block is bound to that processor. A processor read to a block that is 'invalid' (including not present) causes a BusRd transaction to service the miss. The newly loaded block is promoted from invalid to the 'shared' state, whether or not any other cache holds a copy. Any other caches with the block in the 'shared' state observe the BusRd, but take no special action, allowing the memory to respond with the data. However, if a cache has the block in the 'modified' state (there can only be one) and it observes a BusRd transaction on the bus, then it must get involved in the transaction since the memory copy is stale. This cache flushes the data onto the bus, in lieu of memory, and demotes its copy of the block to the shared state. The memory and the requesting cache both pick up the block. This can be accomplished either by a direct cache-to-cache transfer across the bus during this BusRd transaction, or by signalling an error on the BusRd transaction and generating a write transaction to update memory. In the latter case, the original cache will eventually retry its request and obtain the block from memory. (It is also possible to have the flushed data be picked up only by the requesting cache but not by memory, leaving memory still out-of-date, but this requires more states [SwS86]).

Writing into an invalid block is a write miss, which is serviced by first loading the entire block and then modifying the desired bytes within it. The write miss generates a read exclusive bus transaction, which causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the block. The block is raised to the 'modified' state and is then written. If another cache later requests exclusive access, then in response to its Bus-RdX transaction this block will be demoted to the invalid state after flushing the exclusive copy to the bus.

The most interesting transition occurs when writing into a shared block. As discussed earlier this is treated essentially like a write miss, using a read-exclusive bus transaction to acquire exclusive ownership, and we will refer to it as a miss throughout the book. The data that comes back in the read-exclusive can be ignored in this case, unlike when writing to an invalid or not present block, since it is already in the cache. In fact, a common optimization to reduce data traffic in bus protocols is to introduce a new transaction, called a *bus upgrade* or BusUpgr, for this situation. A BusUpgr which obtains exclusive ownership just like a BusRdX, by causing other copies to be invalidated, but it does not return the data for the block to the requestor. Regardless of whether a BusUpgr or a BusRdX is used (let us continue to assume BusRdX), the block transitions to modified state. Additional writes to the block while it is in the modified state generate no additional bus transactions.

A replacement of a block from a cache logically demotes a block to invalid (not present) by removing it from the cache. A replacement therefore causes the state machines for two blocks to change states in that cache: the one being replaced from its current state to invalid, and the one being brought in from invalid to its new state. The latter state change cannot take place before the former, which requires some care in implementation. If the block being replaced was in modified state, the replacement transition from M to I generates a write-back transaction. No special action is taken by the other caches on this transaction. If the block being replaced was in shared or invalid state, then no action is taken on the bus. Replacements are not shown in the state diagram for simplicity.

Note that to specify the protocol completely, for each state we must have outgoing arcs with labels corresponding to all observable events (the inputs from the processor and bus sides), and also actions corresponding to them. Of course, the actions can be null sometimes, and in that case we may either explicitly specify null actions (see states S and M in the figure), or we may simply omit those arcs from the diagram (see state I). Also, since we treat the not-present state as invalid, when a new block is brought into the cache on a miss the state transitions are performed as if the previous state of the block was invalid.

Example 5-6 Using the MSI protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5-3.

Answer The results are shown in Figure 5-14.

Proc. Action	State in P1	State in P2	State in P3	Bus Action	Data Supplied By
1. P1 reads u	S	--	--	BusRd	Memory
2. P3 reads u	S	--	S	BusRd	Memory
3. P3 writes u	I	--	M	BusRdX	Memory
4. P1 reads u	S	--	S	BusRd	P3's cache
5. P2 reads u	S	S	S	BusRd	Memory

Figure 5-14 The 3-state invalidation protocol in action for processor transactions shown in Figure 5-3.

The figure shows the state of the relevant memory block at the end of each transaction, the bus transaction generated, if any, and the entity supplying the data.

With write-back protocols, a block can be written many times before the memory is actually updated. A read may obtain data not from memory but rather from a writer's cache, and in fact it may be this read rather than a replacement that causes memory to be updated. Also, write hits do not appear on the bus, so the concept of "performing a write" is a little different. In fact, it is often simpler to talk, equivalently, about the write being "made visible." A write to a shared or invalid block is made visible by the bus read-exclusive transaction it triggers. The writer will "observe" the data in its cache; other processors will experience a cache miss before observing the write. Write hits to a modified block are visible to other processors, but are observed by them only after a miss through a bus transaction. Formally, in the MSI protocol, the write to a non-modified block is performed or made visible when the BusRdX transaction occurs, and to a modified block when the block is updated in the writer's cache.

Satisfying Coherence

Since both reads and writes to local caches can take place concurrently with the write-back protocol, it is not obvious that it satisfies the conditions for coherence, much less sequential consistency. Let's examine coherence first. The read-exclusive transaction ensures that the writing

cache has the only valid copy when the block is actually written in the cache, just like a write in the write-through protocol. It is followed immediately by the corresponding write being performed in the cache, before any other bus transactions are handled by that cache controller.The only difference from a write-through protocol, with regard to ordering operations to a location, is that not all writes generate bus transactions. However, the key here is that between two transactions for that block that do appear on the bus, only one processor can perform such write hits; this is the processor, say P, that performed the most recent read-exclusive bus transaction w for the block. In the serialization, this write hit sequence therefore appears in program order between w and the next bus transaction for that block. Reads by processor P will clearly see them in this order with respect to other writes. For a read by another processor, there is at least one bus transaction that separates its completion from the completion of these write hits. That bus transaction ensures that this read also sees the writes in the consistent serial order. Thus, reads by all processors see all writes in the same order.

Satisfying Sequential Consistency

To see how sequential consistency is satisfied, let us first appeal to the definition itself and see how a consistent global interleaving of all memory operations may be constructed. The serialization for the bus, in fact, defines a total order on bus transactions for all blocks, not just those to a single block. All cache controllers observe read and read-exclusive transactions in the same order, and perform invalidations in this order. Between consecutive bus transactions, each processor performs a sequence of memory operations (read and writes) in program order. Thus, any execution of a program defines a natural partial order:

> A memory operation M_j is subsequent to operation M_i if (i) the operations are issued by the same processor and M_j follows M_i in program order, or (ii) M_j generates a bus transaction that follows the memory operation for M_i.

This partial order looks graphically like that of Figure 5-6, except the local sequence within a segment has reads and writes and both read-exclusive and read bus transactions play important roles in establishing the orders. Between bus transactions, any interleaving of the sequences from different processors leads to a consistent total order. In a segment between bus transactions, a processor can observe writes by other processors, ordered by previous bus transactions that it generated, as well as its own writes ordered by program order.

We can also see how SC is satisfied in terms of the sufficient conditions, which we will return to when we look further at implementing protocols. Write completion is detected when the read-exclusive bus transaction occurs on the bus and the write is performed in the cache. The third condition, which provides write atomicity, is met because a read either (i) causes a bus transaction that follows that of the write whose value is being returned, in which case the write must have completed globally before the read, or (ii) follows in program order such a read by the same processor, or (iii) follows in program order on the same processor that performed the write, in which case the processor has already waited for the read-exclusive transaction and the write to complete globally. Thus, all the sufficient conditions are easily guaranteed.

Lower-Level Design Choices

To illustrate some of the implicit design choices that have been made in the protocol, let us examine more closely the transition from the M state when a BusRd for that block is observed. In Figure 5-13 we transition to state S and the contents of the memory block are placed on the bus.

While it is imperative that the contents are placed on the bus, it could instead have transitioned to state I. The choice of going to S versus I reflects the designer's assertion that it is more likely that the original processor will continue reading that block than it is that the new processor will soon write to that memory block. Intuitively, this assertion holds for mostly-read data, which is common in many programs. However, a common case where it does not hold is for a flag or buffer that is used to transfer information back and forth between two processes: one processor writes it, the other reads it and modifies it, then the first reads it and modifies it, and so on. The problem with betting on read sharing in this case is that each write is preceded by an invalidate, thereby increasing the latency of the ping-pong operation. Indeed the coherence protocol used in the early Synapse multiprocessor made the alternate choice of directly going from M to I state on a BusRd. Some machines (Sequent Symmetry (model B) and the MIT Alewife) attempt to adapt the protocol when the access pattern indicates such migratory data [CoF93,DDS94]. The implementation of the protocol can also affect the performance of the memory system, as we will see later in the chapter.

5.4.2   A 4-state (MESI) Write-Back Invalidation Protocol

A serious concern with our MSI protocol arises if we consider a sequential application running on a multiprocessor; such multiprogrammed use in fact constitutes the most common workload on small-scale multiprocessors. When it reads in and modifies a data item, in the MSI protocol two bus transactions are generated even though there are never any sharers. The first is a BusRd that gets the memory block in S state, and the second is a BusRdX (or BusUpgr) that converts the block from S to M state. By adding a state that indicates that the block is the only (exclusive) copy but is not modified and loading the block in this state, we can save the latter transaction since the state indicates that no other processor is caching the block. This new state, called exclusive-clean or exclusive-unowned or sometime simply exclusive, indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state it can perform a write and move to modified state without further bus transactions; but it does imply ownership, so unlike in the modified state the cache need not reply upon observing a request for the block (memory has a valid copy). Variants of this MESI protocol are used in many modern microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors. It was first published by researchers at the University of Illinois at Urbana-Champaign [PaP84] and is often referred to as the Illinois protocol [ArB86].

The MESI protocol consists of four states: *modified* (M) or dirty, *exclusive-clean* (E), *shared* (S), and *invalid* (I). I and M have the same semantics as before. E, the exclusive-clean or just exclusive state, means that only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). S means that potentially two or more processors have this block in their cache in an unmodified state.

State Transitions

When the block is first read by a processor, if a valid copy exists in another cache then it enters the processor's cache in the S state as usual. However, if no other cache has a copy at the time (for example, in a sequential application), it enters the cache in the E state. When that block is written by the same processor, it can directly transition from E to M state without generating another bus transaction, since no other cache has a copy. If another cache had obtained a copy in the meantime, the state of the block would have been downgraded from E to S by the snoopy protocol.

This protocol places a new requirement on the physical interconnect of the bus. There must be an additional signal, the shared signal (S), available to the controllers in order to determine on a BusRd if any other cache currently holds the data. During the address phase of the bus transaction, all caches determine if they contain the requested block and, if so, assert the shared signal. This is a wired-OR line, so the controller making the request can observe whether there are any other processors caching the referenced memory block and thereby decide whether to load a requested block in the E state or the S state.

Figure 5-15 shows the state transition diagram for the MESI protocol, still assuming that the BusUpgr transaction is not used. The notation BusRd(S) means that when the bus read transac-
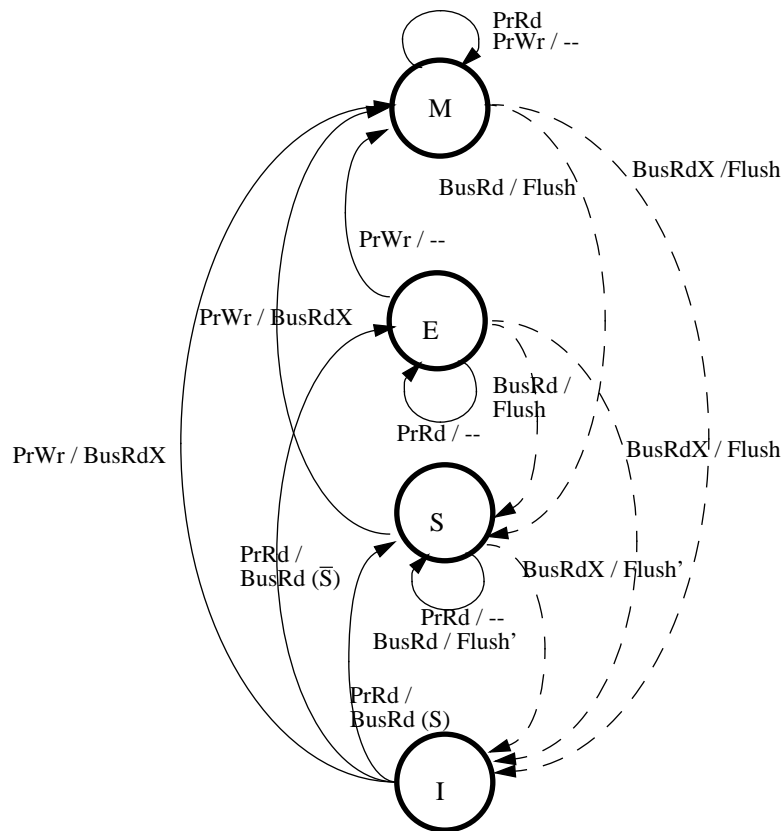
Figure 5-15 State transition diagram for the Illinois MESI protocol.

M, E, S, I stand for the Modified (dirty), Exclusive, Shared and Invalid states respectively. The notation is the same as that in Figure 5-13. The E state helps reduce bus traffic for sequential programs where data are not shared. Whenever feasible, the protocol also makes caches, rather than main memory, supply data for BusRd and BusRdX transactions. Since multiple processors may have a copy of the memory block in their cache, we need to select only one to supply the data on the bus. Flush' is true only for that processor; the remaining processors take null action.

tion occurred the signal "S" was asserted, and BusRd($\overline{S}$) means "S" was unasserted. A plain BusRd means that we don't care about the value of S for that transition. A write to a block in any state will elevate the block to the M state, but if it was in the E state no bus transaction is required. Observing a BusRd will demote a block from E to S, since now another cached copy exists. As usual, observing a BusRd will demote a block from M to S state and cause the block to be flushed on to the bus; here too, the block may be picked up only by the requesting cache and not by main memory, but this will require additional states beyond MESI [SwS86]. Notice that it

is possible for a block to be in the S state even if no other copies exist, since copies may be replaced (S -> I) without notifying other caches. The arguments for coherence and sequential consistency being satisfied here are the same as in the MSI protocol.

Lower-Level Design Choices

An interesting question for the protocol is who should supply the block for a BusRd transaction when both the memory and another cache have a copy of it. In the original Illinois version of the MESI protocol the cache, rather than main memory, supplied the data, a technique called *cache-to-cache sharing*. The argument for this approach was that caches being constructed out of SRAM, rather than DRAM, could supply the data more quickly. However, this advantage is not quite present in many modern bus-based machines, in which intervening in another processor's cache to obtain data is often more expensive than obtaining the data from main memory. Cache-to-cache sharing also adds complexity to a bus-based protocol. Main memory must wait until it is sure that no cache will supply the data before driving the bus, and if the data resides in multiple caches then there needs to be a selection algorithm to determine which one will provide the data. On the other hand, this technique is useful for multiprocessors with physically distributed memory, as we will see in Chapter 6, because the latency to obtain the data from a nearby cache may be much smaller than that for a far away memory unit. This effect can be especially important for machines constructed as a network of SMP nodes, because caches within the SMP node may supply the data. The Stanford DASH multiprocessor [LLJ+92] used such cache-to-cache transfers for this reason.

5.4.3 A 4-state (Dragon) Write-back Update Protocol

We now look at a basic update-based protocol for writeback caches, an enhanced version of which is used in the SUN SparcServer multiprocessors [Cat94]. This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessor system [McC84,TLS88].

The Dragon protocol consists of four states: *exclusive-clean* (E), *shared-clean* (SC), *shared-modified* (SM), and *modified*(M). Exclusive-clean (or exclusive) again means that only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). The motivation for adding the E state in Dragon is the same as that for the MESI protocol. SC means that potentially two or more processors (including this cache) have this block in their cache, and main memory may or may not be up-to-date. SM means that potentially two or more processor's have this block in their cache, main memory is not up-to-date, and it is this processor's responsibility to update the main memory at the time this block is replaced from the cache. A block may be in SM state in only one cache at a time. However, it is quite possible that one cache has the block in SM state while others have it in SC state. Or it may be that no cache has it in SM state but some have it in SC state. This is why when a cache has the block in SC state memory may or may not be up to date; it depends on whether some cache has it in SM state. M means, as before, that the block is modified (dirty) in this cache alone, main memory is stale, and it is this processor's responsibility to update main memory on replacement. Note that there is no explicit invalid (I) state as in the previous protocols. This is because Dragon is an update-based protocol; the protocol always keeps the blocks in the cache up-to-date, so it is always okay to use the data present in the cache. However, if a block is not present in a cache at all, it can be imagined to be there in a special invalid or not-present state.[1]

The processor requests, bus transactions, and actions for the Dragon protocol are similar to the Illinois MESI protocol. The processor is still assumed to issue only read (PrRd) and write (PrWr) requests. However, given that we do not have an invalid state in the protocol, to specify actions when a new memory block is first requested by the processor, we add two more request types: processor read-miss (PrRdMiss) and write-miss (PrWrMiss). As for bus transactions, we have bus read (BusRd), bus update (BusUpd), and bus writeback (BusWB). The BusRd and BusWB transactions have the usual semantics as defined for the earlier protocols. BusUpd is a new transaction that takes the specific word written by the processor and broadcasts it on the bus so that all other processors' caches can update themselves. By only broadcasting the contents of the specific word modified rather than the whole cache block, it is hoped that the bus bandwidth is more efficiently utilized. (See Exercise for reasons why this may not always be the case.) As in the MESI protocol, to support the E state there is a shared signal (S) available to the cache controller. This signal is asserted if there are any processors, other than the requestor, currently caching the referenced memory block. Finally, as for actions, the only new capability needed is for the cache controller to update a locally cached memory block (labeled Update) with the contents that are being broadcast on the bus by a relevant BusUpd transaction.
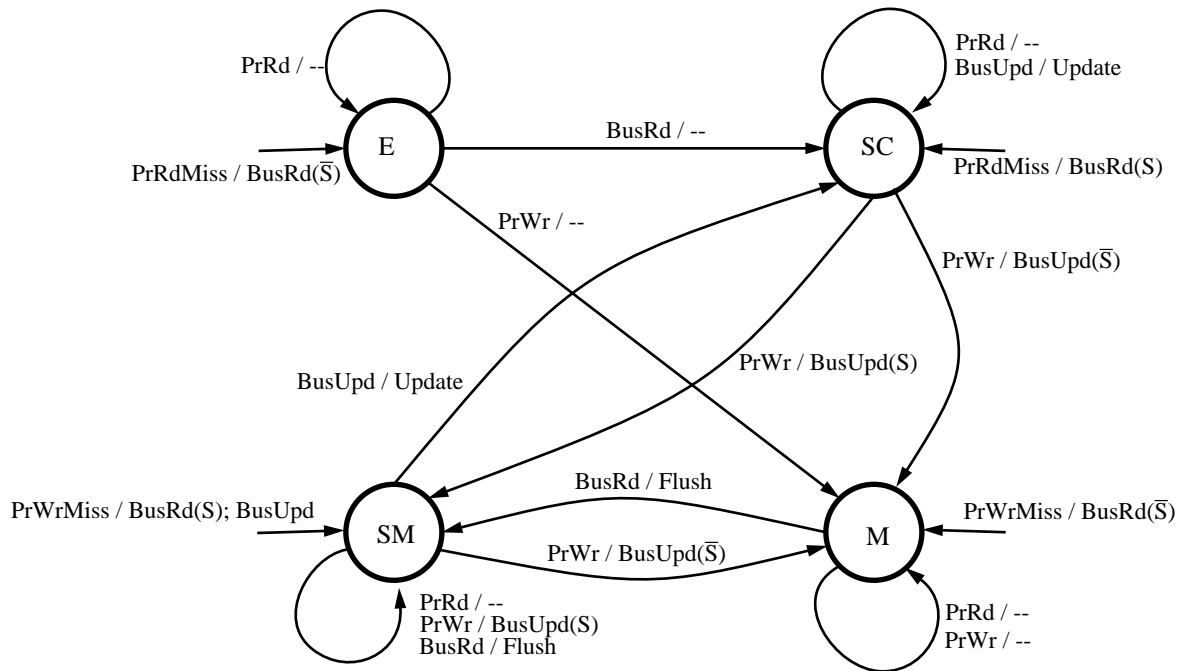
Figure 5-16 State-transition diagram for the Dragon update protocol.

The four states are valid-exclusive (E), shared-clean (SC), shared-modified (SM), and modified (M). There is no invalid (I) state because the update protocol always keeps blocks in the cache up-to-date.

---

1. Logically there is another state as well, but it is rather crude. A mode bit is provided with each block to force a miss when the block is accessed. Initialization software reads data into every line in the cache with the miss mode turned on, to ensure that the processor will miss the first time it references a block that maps to that line.

State Transitions

Figure 5-16 shows the state transition diagram for the Dragon update protocol. To take a processor-centric view, one can also explain the diagram in terms of actions taken when a processor incurs a read-miss, a write-hit, or a write-miss (no action is ever taken on a read-hit).

Read Miss: A BusRd transaction is generated. Depending on the status of the shared-signal (S), the block is loaded in the E or SC state in the local cache. More specifically, if the block is in M or SM states in one of the other caches, that cache asserts the shared signal and supplies the latest data for that block on the bus, and the block is loaded in local cache in SC state (it is also updated in main memory; if we did not want to do this, we would need more states). If the other cache had it in state M, it changes its state to SM. If no other cache has a copy, then the shared line remains unasserted, the data are supplied by the main memory and the block is loaded in local cache in E state.

Write: If the block is in the SM or M states in the local cache, then no action needs to be taken. If the block is in the E state in the local cache, then it internally changes to M state and again no further action is needed. If the block is in SC state, however, a BusUpd transaction is generated. If any other caches have a copy of the data, they update their cached copies, and change their state to SC. The local cache also updates its copy of the block and changes its state to SM. If no other cache has a copy of the data, the shared-signal remains unasserted, the local copy is updated and the state is changed to M. Finally, if on a write the block is not present in the cache, the write is treated simply as a read-miss transaction followed by a write transaction. Thus first a BusRd is generated, and then if the block was also found in other caches (i.e., the block is loaded locally in the SC state), a BusUpd is generated.

Replacement: On a replacement (arcs not shown in the figure), the block is written back to memory using a bus transaction only if it is in the M or SM state. If it is in the SC state, then either some other cache has it in SM state, or noone does in which case it is already valid in main memory.

Example 5-7   Using the Dragon update protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5-3.

Answer   The results are shown in Figure 5-17. We can see that while for processor actions 3 and 4 only one word is transferred on the bus in the update protocol, the whole memory block is transferred twice in the invalidation-based protocol. Of course, it is easy to construct scenarios where the invalidation protocol does much better than the update protocol, and we will discuss the detailed tradeoffs later in Section 5.5.

Lower-Level Design Choices

Again, many implicit design choices have been made in this protocol. For example, it is feasible to eliminate the shared-modified state. In fact, the update-protocol used in the DEC Firefly multiprocessor did exactly that. The reasoning was that every time the BusUpd transaction occurs, the main memory can also update its contents along with the other caches holding that block, and therefore, a shared-modified state is not needed. The Dragon protocol was instead based on the assumption that since caches use SRAM and main memory uses DRAM, it is much quicker to update the caches than main memory, and therefore it is inappropriate to wait for main memory

Proc. Action	State in P1	State in P2	State in P3	Bus Action	Data Supplied By
1. P1 reads u	E	--	--	BusRd	Memory
2. P3 reads u	SC	--	SC	BusRd	Memory
3. P3 writes u	SC	--	SM	BusUpd	P3
4. P1 reads u	SC	--	SM	null	--
5. P2 reads u	SC	SC	SM	BusRd	P3

Figure 5-17 The Dragon update protocol in action for processor transactions shown in Figure 5-3.

The figure shows the state of the relevant memory block at the end of each transaction, the bus transaction generated, if any, and the entity supplying the data.

to be updated on all BusUpd transactions. Another subtle choice, for example, relates to the action taken on cache replacements. When a shared-clean block is replaced, should other caches be informed of that replacement via a bus-transaction, so that if there is only one remaining cache with a copy of the memory block then it can change its state to valid-exclusive or modified? The advantage of doing this would be that the bus transaction upon the replacement may not be in the critical path of a memory operation, while the later bus transaction it saves might be.

Since all writes appear on the bus in an update protocol, write serialization, write completion detection, and write atomicity are all quite straightforward with a simple atomic bus, a lot like they were in the write-through case. However, with both invalidation and update based protocols, there are many subtle implementation issues and race conditions that we must address, even with an atomic bus and a single-level cache. We will discuss these in Chapter 6, as well as more realistic scenarios with pipelined buses, multi-level cache hierarchies, and hardware techniques that can reorder the completion of memory operations like write buffers. Nonetheless, we can quantitatively examine protocol tradeoffs at the state diagram level that we have been considering so far.

5.5 Assessing Protocol Design Tradeoffs

Like any other complex system, the design of a multiprocessor requires many decisions to be made. Even when a processor has been picked, the designer must decide on the maximum number of processors to be supported by the system, various parameters of the cache hierarchy (e.g., number of levels in the hierarchy, and for each level the cache size, associativity, block size, and whether the cache is write-through or writeback), the design of the bus (e.g., width of the data and address buses, the bus protocol), the design of the memory system (e.g., interleaved memory banks or not, width of memory banks, size of internal buffers), and the design of the I/O subsystem. Many of the issues are similar to those in uniprocessors [Smi82], but accentuated. For example, a write-through cache standing before the bus may be a poor choice for multiprocessors because the bus bandwidth is shared by many processors, and memory may need to be interleaved more because it services cache misses from multiple processors. Greater cache associativity may also be useful in reducing conflict misses that generate bus traffic.

A crucial new design issue for a multiprocessor is the cache coherence protocol. This includes protocol class (invalidation or update), protocol states and actions, and lower-level implementation tradeoffs that we will examine later. Protocol decisions interact with all the other design issues. On one hand, the protocol influences the extent to which the latency and bandwidth characteristics of system components are stressed. On the other, the performance characteristics as well as organization of the memory and communication architecture influence the choice of protocols. As discussed in Chapter 4, these design decisions need to be evaluated relative to the behavior of real programs. Such evaluation was very common in the late 1980s, albeit using an immature set of parallel programs as workloads [ArB86,AgG88,EgK88,EgK89a,EgK89b].

Making design decisions in real systems is part art and part science. The art is the past experience, intuition, and aesthetics of the designers, and the science is workload-driven evaluation. The goals are usually to meet a cost-performance target and to have a balanced system so that no individual resource is a performance bottleneck yet each resource has only minimal excess capacity. This section illustrates some key protocol tradeoffs by putting the workload driven evaluation methodology from Chapter 4 to action.

The basic strategy is as follows. The workload is executed on a simulator of a multiprocessor architecture, which has two coupled parts: a reference generator and a memory system simulator. The reference generator simulates the application program's processes and issues memory references to the simulator. The simulator simulates the operation of the caches, bus and state machines, and returns timing information to the reference generator, which uses this information to determine from which process to issue the next reference. The reference generator interleaves references from different processes, preserving timing relationships from the simulator as well as the synchronization relationships in the parallel program. By observing the state transitions in the simulator, we can determine the frequency of various events such as cache misses and bus transactions. We can then evaluate the effect of protocol choices in terms of other design parameters such as latency and bandwidth requirements.

Choosing parameters according to the methodology of Chapter 4, this section first establishes the basic state transition characteristics generated by the set of applications for the 4-state, Illinois MESI protocol. It then illustrates how to use these frequency measurements to obtain a preliminary quantitative analysis of design tradeoffs raised by the example protocols above, such as the use of the fourth, exclusive state in the MESI protocol and the use of BusUpgr rather then Bus-RdX transactions for the S->M transition. It also illustrates more traditional design issues, such as how the cache block size—the granularity of both coherence and communication—impacts the latency and bandwidth needs of the applications. To understand this effect, we classify cache misses into categories such as cold, capacity, and sharing misses, examine the effect of block size on each, and explain the results in light of application characteristics. Finally, this understanding of the applications is used to illustrate the tradeoffs between invalidation-based and update-based protocols, again in light of latency and bandwidth implications.

One important note is that this analysis is based on the frequency of various important events, not the absolute times taken or therefore the performance. This approach is common in studies of cache architecture, because the results transcend particular system implementations and technology assumptions. However, it should be viewed as only a preliminary analysis, since many detailed factors that might affect the performance tradeoffs in real systems are abstracted away. For example, measuring state transitions provides a means of calculating miss rates and bus traffic, but realistic values for latency, overhead, and occupancy are needed to translate the rates into the actual bandwidth requirements imposed on the system. And the bandwidth requirements

themselves do not translate into performance directly, but only indirectly by increasing the cost of misses due to contention. To obtain an estimate of bandwidth requirements, we may artificially assume that every reference takes a fixed number of cycles to complete. The difficulty is incorporating contention, because it depends on the timing parameters used and on the burstiness of the traffic which is not captured by the frequency measurements.

The simulations used in this section do not model contention, and in fact they do not even pretend to assume a realistic cost model. All memory operations are assumed to complete in the same amount of time (here a single cycle) regardless of whether they hit or miss in the cache. There are Three main reasons for this. First, the focus is on understanding inherent protocol behavior and tradeoffs in terms of event frequencies, not so much on performance. Second, since we are experimenting with different cache block sizes and organizations, we would like the interleaving of references from application processes on the simulator to be the same regardless of these choices; i.e. all protocols and block sizes should see the same trace of references. With execution-driven simulation, this is only possible if we make the cost of every memory operation the same in the simulations (e.g., one cycle), whether it hits or misses. Otherwise, if a reference misses with a small cache block but hits with a larger one (for example) then it will be delayed by different amounts in the interleaving in the two cases. It would therefore be difficult to determine which effects are inherently due to the protocol and which are due to the particular parameter values chosen. The third reason is that realistic simulations that model contention would take a lot more time. The disadvantage of using this simple model is that the timing model may affect some of the frequencies we observe; however, this effect is small for the applications we study.

5.5.1 Workloads

The illustrative workloads for coherent shared address space architectures include six parallel applications and computational kernels and one multiprogrammed workload. The parallel programs run in batch mode with exclusive access to the machine and do not include operating system activity, at least in the simulations, while the multiprogrammed workload includes operating system activity. The number of applications we use is relatively small, but they are primarily for illustration and we try to choose programs that represent important classes of computation and have widely varying characteristics.

The parallel applications used here are taken from the SPLASH2 application suite (see Appendix A) and were described in previous chapters. Three of them (Ocean, Barnes-Hut, and Raytrace) were used as case studies for developing parallel programs in Chapters 2 and 3. The frequencies of basic operations for the applications is given in Table 4-1. We now study them in more detail to assess design tradeoffs in cache coherency protocols.

Bandwidth requirement under the MESI protocol

Driving the address traces for the workloads depicted in Table 4-1 through a cache simulator modeling the Illinois MESI protocol generates the state-transition frequencies in Table 5-1. The data are presented as number of state-transitions of a particular type per 100 references issued by the processors. Note that in the tables, a new state NP (not-present) is introduced. This addition helps to clarify transitions where on a cache miss, one block is replaced (creating a transition from one of I, E, S, or M to NP) and a new block is brought in (creating a transition from NP to one of I, E, S, or M). For example, we can distinguish between writebacks done due to replacements (M -> NP transitions) and writebacks done because some other processor wrote to a modi-

fied block that was cached locally (M -> I transitions). Note that the sum of state transitions can be greater than 100, even though we are presenting averages per 100 references, because some references cause multiple state transitions. For example, a write-miss can cause two-transitions in the local processor's cache (e.g., S -> NP for the old block, and NP -> M) for the incoming block, plus transitions in other processor's cache invalidating their copies (I/E/S/M -> I).[1] This low-level state-transition frequency data is an extremely useful way to answer many different kinds of "what-if" questions. As a first example, we can determine the bandwidth requirement these applications would place on the memory system.

Table 5-1 State transitions per 1000 data memory references issued by the applications. The data assumes 16 processors, 1 Mbyte 4-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

Appln.	From/To	NP	I	E	S	M
Barnes-Hut	NP	0	0	0.0011	0.0362	0.0035
	I	0.0201	0	0.0001	0.1856	0.0010
	E	0.0000	0.0000	0.0153	0.0002	0.0010
	S	0.0029	0.2130	0	97.1712	0.1253
	M	0.0013	0.0010	0	0.1277	902.782
LU	NP	0	0	0.0000	0.6593	0.0011
	I	0.0000	0	0	0.0002	0.0003
	E	0.0000	0	0.4454	0.0004	0.2164
	S	0.0339	0.0001	0	302.702	0.0000
	M	0.0001	0.0007	0	0.2164	697.129
Ocean	NP	0	0	1.2484	0.9565	1.6787
	I	0.6362	0	0	1.8676	0.0015
	E	0.2040	0	14.0040	0.0240	0.9955
	S	0.4175	2.4994	0	134.716	2.2392
	M	2.6259	0.0015	0	2.2996	843.565
Radiosity	NP	0	0	0.0068	0.2581	0.0354
	I	0.0262	0	0	0.5766	0.0324
	E	0	0.0003	0.0241	0.0001	0.0060
	S	0.0092	0.7264	0	162.569	0.2768
	M	0.0219	0.0305	0	0.3125	839.507
	NP	0	0	0.0030	1.3153	5.4051

---

1. For the Multiprog workload, to speed up the simulations a 32Kbyte instruction cache is used as a filter, before passing the instruction references to the 1 Mbyte unified instruction and data cache. The state transition frequencies for the instruction references are computed based only on those references that missed in the L1 instruction cache. This filtering does not affect any of the bus-traffic data that we will generate using these numbers. Also, for Multiprog we present data separately for kernel instructions, kernel data references, user instructions, and user data references. A given reference may produce transitions of multiple types. For example, if a kernel instruction miss causes a modified user-data-block to be written back, then we will have one transition for kernel instructions from NP -> E/S and another transition for user data reference category from M -> NP.

Table 5-1 State transitions per 1000 data memory references issued by the applications. The
data assumes 16 processors, 1 Mbyte 4-way set-associative caches, 64-byte cache blocks, and the
Illinois MESI coherence protocol.

Appln.	From/To	NP	I	E	S	M
	I	0.0485	0	0	0.4119	1.7050
Radix	E	0.0006	0.0008	0.0284	0.0001	0
	S	0.0109	0.4156	0	84.6671	0.3051
	M	0.0173	4.2886	0	1.4982	906.945
	NP	0	0	1.3358	0.15486	0.0026
	I	0.0242	0	0.0000	0.3403	0.0000
Raytrace	E	0.8663	0	29.0187	0.3639	0.0175
	S	1.1181	0.3740	0	310.949	0.2898
	M	0.0559	0.0001	0	0.2970	661.011
	NP	0	0	0.1675	0.5253	0.1843
Multiprog	I	0.2619	0	0.0007	0.0072	0.0013
User	E	0.0729	0.0008	11.6629	0.0221	0.0680
Data	S	0.3062	0.2787	0	214.6523	0.2570
References	M	0.2134	0.1196	0	0.3732	772.7819
	NP	0	0	3.2709	15.7722	0
Multiprog	I	0	0	0	0	0
User	E	1.3029	0	46.7898	1.8961	0
Instruction	S	16.9032	0	0	981.2618	0
References	M	0	0	0	0	0
	NP	0	0	1.0241	1.7209	4.0793
Multiprog	I	1.3950	0	0.0079	1.1495	0.1153
Kernel	E	0.5511	0.0063	55.7680	0.0999	0.3352
Data	S	1.2740	2.0514	0	393.5066	1.7800
References	M	3.1827	0.3551	0	2.0732	542.4318
	NP	0	0	2.1799	26.5124	0
Multiprog	I	0	0	0	0	0
Kernel	E	0.8829	0	5.2156	1.2223	0
Instruction	S	24.6963	0	0	1075.2158	0
References	M	0	0	0	0	0

Example 5-8   Suppose that the integer-intensive applications run at 200 MIPS per processor, and
the floating-point intensive applications at 200 MFLOPS per processor. Assuming
that cache block transfers move 64 bytes on the data lines and that each bus

transaction involves six bytes of command and address on the address lines, what is the traffic generated per processor?

Answer The first step is to calculate the amount of traffic per instruction. We determine what bus-action needs to be taken for each of the possible state-transitions and how much traffic is associated with each transaction. For example, a M --> NP transition indicates that due to a miss a cache block needs to be written back. Similarly, an S -> M transition indicates that an upgrade request must be issued on the bus. The bus actions corresponding to all possible transitions are shown in Table 5-2. All transactions generate six bytes of address traffic and 64 bytes of address traffic, except BusUpgr, which only generates address traffic.

Table 5-2 Bus actions corresponding to state transitions in Illinois MESI protocol.

From/To	NP	I	E	S	M
NP	--	--	BusRd	BusRd	BusRdX
I	--	--	BusRd	BusRd	BusRdX
E	--	--	--	--	--
S	--	--	not poss.	--	BusUpgr
M	BusWB	BusWB	not poss.	BusWB	--

At this point in the analysis the cache parameters and cache protocol are pinned down (they are implicit in the state-transition data since they were provided to the simulator in order to generate the data.) and the bus design parameters are pinned down as well. We can now compute the traffic generated by the processors. Using Table 5-2 we can convert the transitions per 1000 memory references in Table 5-1 to bus transactions per 1000 memory references, and convert this to address and data traffic by multiplying by the traffic per transaction. Using the frequency of memory accesses in Table 4-1 we convert this to traffic per instruction, or per MFLOPS. Finally, multiplying by the assuming processing rate, we get the address and data bandwidth requirement for each application. The result of this calculation is shown by left-most bar in Figure 5-18.

The calculation in the example above gives the average bandwidth requirement under the assumption that the bus bandwidth is enough to allow the processors to execute at full rate. In practice. This calculation provides a useful basis for sizing the system. For example, on a machine such as the SGI Challenge with 1.2 GB/s of data bandwidth, the bus provides sufficient bandwidth to support 16 processors on the applications other than Radix. A typical rule of thumb is to leave 50% "head room" to allow for burstiness of data transfers. If the Ocean and Multiprog workloads were excluded, the bus could support up to 32 processors. If the bandwidth is not sufficient to support the application, the application will slow down due to memory waits. Thus, we would expect the speedup curve for Radix to flatten out quite quickly as the number of processors grows. In general a multiprocessor is used for a variety of workloads, many with low per-processor bandwidth requirements, so the designer will choose to support configurations of a size that would overcommit the bus on the most demanding applications.

5.5.2 Impact of Protocol Optimizations

Given this base design point, we can evaluate protocol tradeoffs under common machine parameter assumptions.

Example 5-9   We have described two invalidation protocols in this chapter -- the basic 3-state invalidation protocol and the Illinois MESI protocol. The key difference between them is the existence of the valid-exclusive state in the latter. How much bandwidth savings does the E state buy us?

Answer   The main advantage of the E state is that when going from E->M, no traffic need be generated. In this case, in the 3-state protocol we will have to generate a BusUpgr transaction to acquire exclusive ownership for that memory block. To compute bandwidth savings, all we have to do is put a BusUpgr for the E->M transition in Table 5-2, and then recompute the traffic as before. The middle bar in Figure 5-18 shows the resulting bandwidth requirements.

The example above illustrates how anecdotal rationale for a more complex design may not stand up to quantitative analysis. We see that, somewhat contrary to expectations, the E state offers negligible savings. This is true even for the Multiprog workload, which consists primarily of sequential jobs. The primary reason for this negligible gain is that the fraction of E->M transitions is quite small. In addition, the BusUpgr transaction that would have been needed for the S->M transition takes only 6 bytes of address traffic and no data traffic.

Example 5-10   Recall that for the 3-state protocol, for a write that finds the memory block in shared state in the cache we issue a BusUpgr request on the bus, rather than a BusRdX. This saves bandwidth, as no data need be transferred for a BusUpgr, but it does complicate the implementation, since local copy of data can be invalidated in the cache before the upgrade request comes back. The question is how much bandwidth are we saving for taking on this extra complexity.

Answer   To compute the bandwidth for the less complex implementation, all we have to do is put in BusRdX in the E->M and S->M transitions in Table 5-2 and then recompute the bandwidth numbers. The results for all applications are shown in the right-most bar in Figure 5-18. While for most applications the difference in bandwidth is small, Ocean and Multiprog kernel-data references show that it can be as large as 10-20% on demanding applications.

The performance impact of these differences in bandwidth requirement depend on how the bus transactions are actually implemented. However, this high level analysis indicates to the designer where more detailed evaluation is required.

Finally, as we had discussed in Chapter 3, given the input data-set sizes we are using for the above applications, it is important that we run the Ocean, Raytrace, Radix applications for smaller 64 Kbyte cache sizes. The raw state-transition data for this case are presented in Table 5-3 below, and the per-processor bandwidth requirements are shown in Figure 5-19. As we can see,
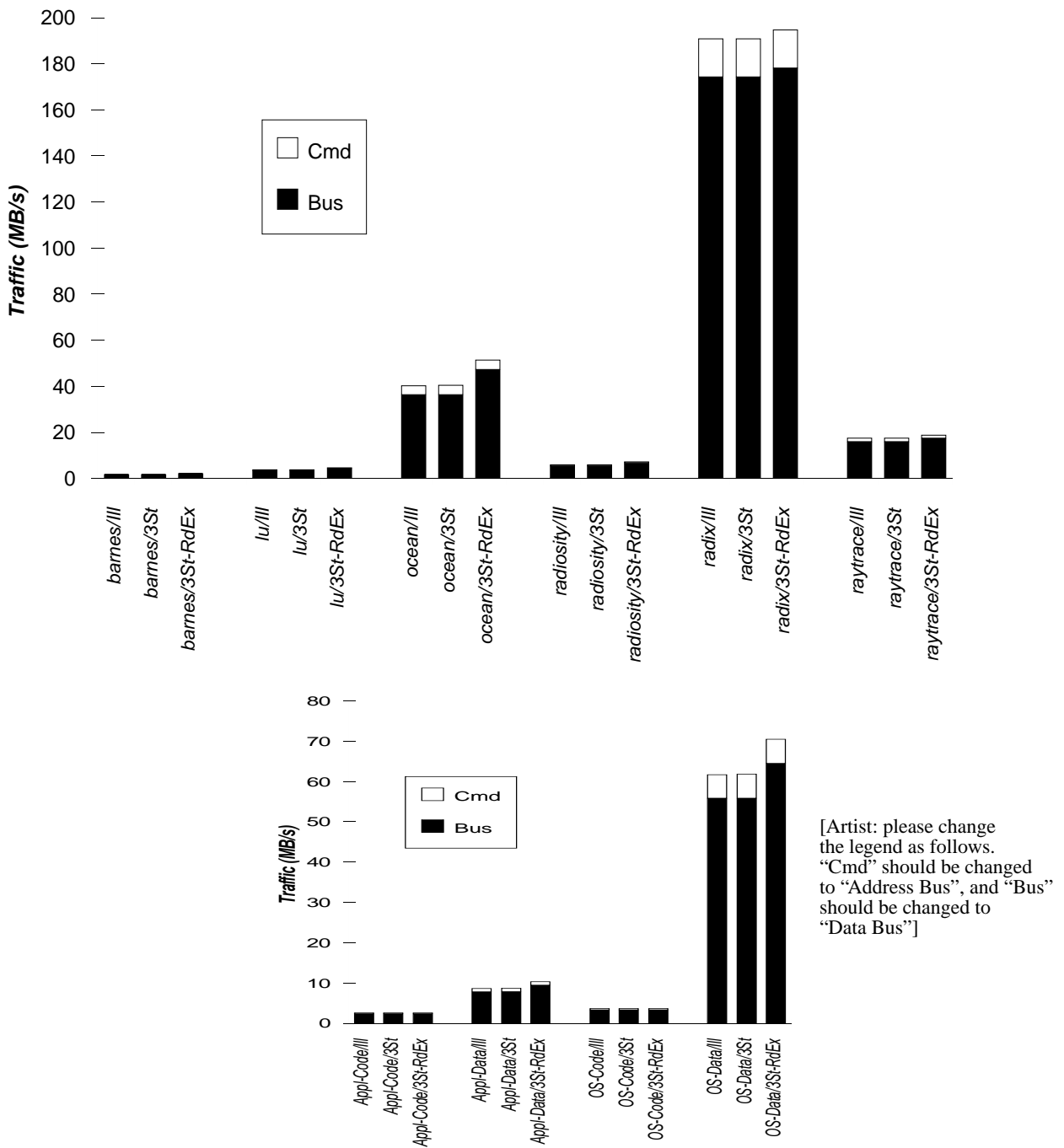
Figure 5-18 Per processor bandwidth requirements for the various applications assuming 200MIPS/MFLOPS processors.

The top bar-chart shows data for SPLASH-2 applications and the bottom chart data for the Multiprog workload. The traffic is split into data traffic and address+command bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic 3-state invalidation protocol without the E state as described in Section 5.4.1, and the right most bar shows the traffic for the 3-state protocol when we use BusRdX instead of BusUpgr for S -> M transitions.

Table 5-3 State transitions per 1000 memory references issued by the applications. The data assumes 16 processors, 64 Kbyte 4-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

Appln.	From/To	NP	I	E	S	M
Ocean	NP	0	0	26.2491	2.6030	15.1459
	I	1.3305	0	0	0.3012	0.0008
	E	21.1804	0.2976	452.580	0.4489	4.3216
	S	2.4632	1.3333	0	113.257	1.1112
	M	19.0240	0.0015	0	1.5543	387.780
Radix	NP	0	0	3.5130	0.9580	11.3543
	I	1.6323	0	0.0001	0.0584	0.5556
	E	3.0299	0.0005	52.4198	0.0041	0.0481
	S	1.4251	0.1797	0	56.5313	0.1812
	M	8.5830	2.1011	0	0.7695	875.227
Raytrace	NP	0	0	7.2642	3.9742	0.1305
	I	0.0526	0	0.0003	0.2799	0.0000
	E	6.4119	0	131.944	0.7973	0.0496
	S	4.6768	0.3329	0	205.994	0.2835
	M	0.1812	0.0001	0	0.2837	660.753

not having one of the critical working sets fit in the processor cache can dramatically increase the bandwidth required. A 1.2 Gbyte/sec bus can now barely support 4 processors for Ocean and Radix, and 16 processors for Raytrace.

5.5.3 Tradeoffs in Cache Block Size

The cache organization is a critical performance factor of all modern computers, but it is especially so in multiprocessors. In the uniprocessor context, cache misses are typically categorized into the "three-Cs": compulsory, capacity, and conflict misses [HiS89,PH90]. Many studies have examined how cache size, associativity, and block size affect each category of miss. *Compulsory misses*, or *cold misses*, occur on the first reference to a memory block by a processor. *Capacity misses* occur when all the blocks that are referenced by a processor during the execution of a program do not fit in the cache (even with full associativity), so some blocks are replaced and later accessed again. *Conflict* or *collision misses* occur in caches with less than full associativity, when the collection of blocks referenced by a program that map to a single cache set do not fit in the set. They are misses that would not have occurred in a fully associative cache.

Capacity misses are reduced by enlarging the cache. Conflict misses are reduced by increasing the associativity or increasing the number of blocks (increasing cache size or reducing block size). Cold misses can only be reduced by increasing the block size, so that a single cold miss will bring in more data that may be accessed as well. What makes cache design challenging is that these factors trade-off against one another. For example, increasing the block size for a fixed cache capacity will reduce the number of blocks, so the reduced cold misses may come at the cost of increased conflict misses. In addition, variations in cache organization can affect the miss penalty or the hit time, and therefore cycle time.
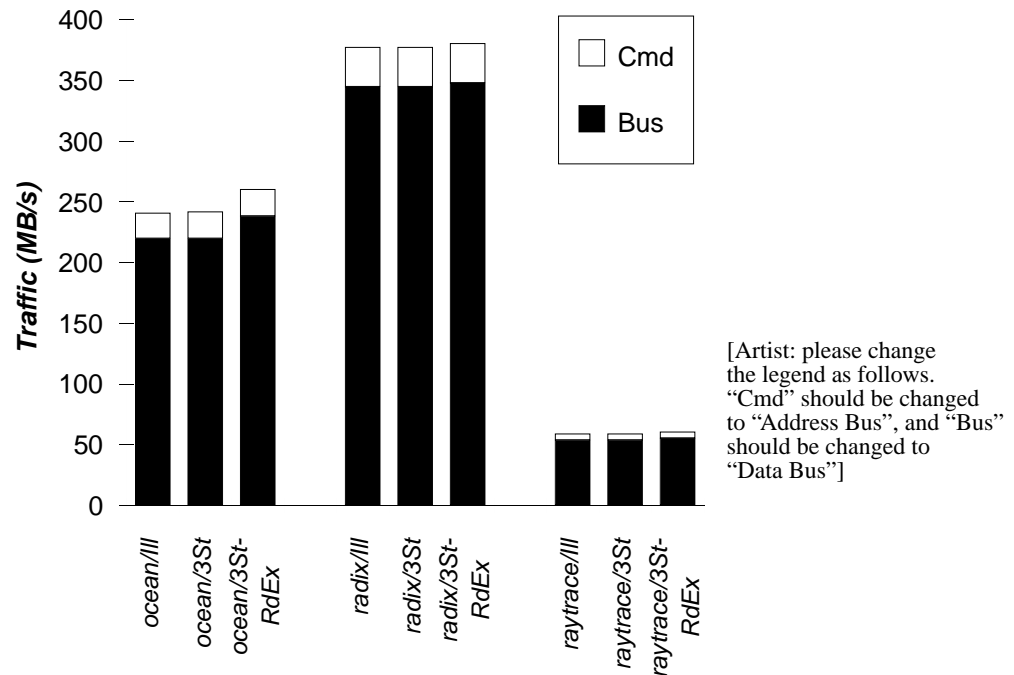
Figure 5-19 Per processor bandwidth requirements for the various applications assuming 200MIPS/MFLOPS processors and 64 Kbyte caches.

The traffic is split into data traffic and address+cmd bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic 3-state invalidation protocol without the E state as described in Section 5.4.1, and the right most bar shows the traffic for the 3-state protocol when we use BusRdX instead of BusUpgr for S -> M transitions.

Cache-coherent multiprocessors introduce a fourth category of misses: *coherence misses*. These occur when blocks of data are shared among multiple caches, and are of two types: true sharing and false sharing. True sharing occurs when a data word produced by one processor is used by another. False sharing occurs when independent data words for different processors happen to be placed in the same block. The cache block size is the granularity (or unit) of fetching data from the main memory, and it is typically also used as the granularity of coherence. That is, on a write by a processor, the whole cache block is invalidated in other processors' caches. A *true sharing miss* occurs when one processor writes some words in a cache block, invalidating that block in another processor's cache, and then the second processor reads one of the modified words. It is called a "true" sharing miss because the miss truly communicates newly defined data values that are used by the second processor; such misses are "essential" to the correctness of the program in an invalidation-based coherence protocol, regardless of interactions with the machine organization. On the other hand, when one processor writes some words in a cache block and then another processor reads (or writes) different words in the same cache block, the invalidation of the block and subsequent cache miss occurs as well, even though no useful values are being communicated between the processors. These misses are thus called *false-sharing misses* [DSR+93]. As cache block size is increased, the probability of distinct variables being accessed by different processors but residing on the same cache block increases. Technology pushes in the direction of large cache block sizes (e.g., DRAM organization and access modes, and the need to obtain high-band-

width data transfers by amortizing overhead), so it is important to understand the potential impact of false sharing misses and how they may be avoided.

True-sharing misses are inherent to a given parallel decomposition and assignment; like cold misses, the only way to decrease them is by increasing the block size and increasing spatial locality of communicated data. False sharing misses, on the other hand, are an example of the artifactual communication discussed in Chapter 3, since they are caused by interactions with the architecture. In contrast to true sharing and cold misses, false sharing misses can be decreased by reducing the cache block size, as well as by a host of other optimizations in software (orchestration) and hardware. Thus, there is a fundamental tension in determining the best cache block size, which can only be resolved by evaluating the options against real programs.

A Classification of Cache Misses

The flowchart in Figure 5-20 gives a detailed algorithm for classification of misses.[1] Understanding the details is not very important for now—it is enough for the rest of the chapter to understand the definitions above—but it adds insight and is a useful exercise. In the algorithm, we define the *lifetime* of a block as the time interval during which the block remains valid in the cache, that is, from the occurrence of the miss until its invalidation, replacement, or until the end of simulation. Observe that we cannot classify a cache miss when it occurs, but only when the fetched memory block is replaced or invalidated in the cache, because we need to know if during the block's lifetime the processor used any words that were written since the last true-sharing or essential miss. Let us consider the simple cases first. Cases 1 and 2 are straightforward cold misses occurring on previously unwritten blocks. Cases 7 and 8 reflect false and true sharing on a block that was previously invalidated in the cache, but not discarded. The type of sharing is determined by whether the word(s) modified since the invalidation are actually used. Case 11 is a straightforward capacity (or conflict) miss, since the block was previously replaced from the cache and the words in the block have not been accessed since last modified. All of the other cases refer to misses that occur due to a combination of factors. For example, cases 4 and 5 are cold misses because this processor has never accessed the block before, however, some other processor has written the block, so there is also sharing (false or true, depending on which words are accessed). Similarly, we can have sharing (false or true) on blocks that were previously discarded due to capacity. Solving only one of the problems may not necessarily eliminate such misses. Thus, for example, if a miss occurs due to both false-sharing and capacity problems, then eliminating the false-sharing problem by reducing block size will likely not eliminate that miss. On the other hand, sharing misses are in some sense more fundamental than capacity misses, since they will remain there even if the size of cache is increased to infinity.

Example 5-11   To illustrate the definitions that we have just given, Table 5-4 below shows how to classify references issued by three processors P1, P2, and P3. For this example, we

---

1. In this classification we do not distinguish conflict from capacity misses, since they both are a result of the available resources becoming full and the difference between them (set versus entire cache) does not shed additional light on multiprocessor issues.

assume that each processor's cache consists of a single 4-word cache block. We also assume that the caches are all initially empty.

Table 5-4 Classifying misses in an example reference stream from 3 processors. If multiple references are listed in the same row, we assume that P1 issues before P2 and P2 issues before P3. The notation ld/st wi refers to load/store of word i. The notation Pi.j points to the memory reference issued by processor i at row j.

Seq.	P1	P2	P3	Miss Classification
1	ld w0		ld w2	P1 and P3 miss; but we will classify later on replace/inval
2			st w2	P1.1: pure-cold miss; P3.2: upgrade
3		ld w1		P2 misses, but we will classify later on replace/inval
4		ld w2	ld w7	P2 hits; P3 misses; P3.1: cold miss
5	ld w5			P1 misses
6		ld w6		P2 misses; P2.3: cold-true-sharing miss (w2 accessed)
7		st w6		P1.5: cold miss; p2.7: upgrade; P3.4: pure-cold miss
8	ld w5			P1 misses
9	ld w6		ld w2	P1 hits; P3 misses
10	ld w2	ld w1		P1, P2 miss; P1.8: pure-true-share miss; P2.6: cold miss
11	st w5			P1 misses; P1.10: pure-true-sharing miss
12			st w2	P2.10: capacity miss; P3.11: upgrade
13			ld w7	P3 misses; P3.9: capacity miss
14			ld w2	P3 misses; P3.13: inval-cap-false-sharing miss
15	ld w0			P1 misses; P1.11: capacity miss

Impact of Block Size on Miss Rate

Applying the classification algorithm of Figure 5-20 to simulated runs of a workload we can determine how frequently the various kinds of misses occur in programs and how the frequencies change with variations in cache organization, such as block size. Figure 5-21 shows the decomposition of the misses for the example applications running on 16 processors, 1 Mbyte 4-way set associative caches, as the cache block size is varied from 8 bytes to 256 bytes. The bars show the four basic types of misses, cold misses (cases 1 and 2), capacity misses (case 11), true-sharing misses, (cases 4, 6, 8, 10, 12), and false-sharing misses (cases 3, 5, 7, and 9). In addition, the figure shows the frequency of *upgrades,* which are writes that find the block in the cache but in shared state. They are different than the other types of misses in that the cache already has the valid data so all that needs to be acquired is exclusive ownership, and they are are not included in the classification scheme of Figure 5-20. However, they are still usually considered to be misses since they generate traffic on the interconnect and can stall the processor.

While the table only shows data for the default data-sets, in practice it is very important to examine the results as input data-set size and number of processors are scaled, before drawing conclusions about the false-sharing or spatial locality of an application. The impact of such scaling is elaborated briefly in the discussion below.

Miss Classification
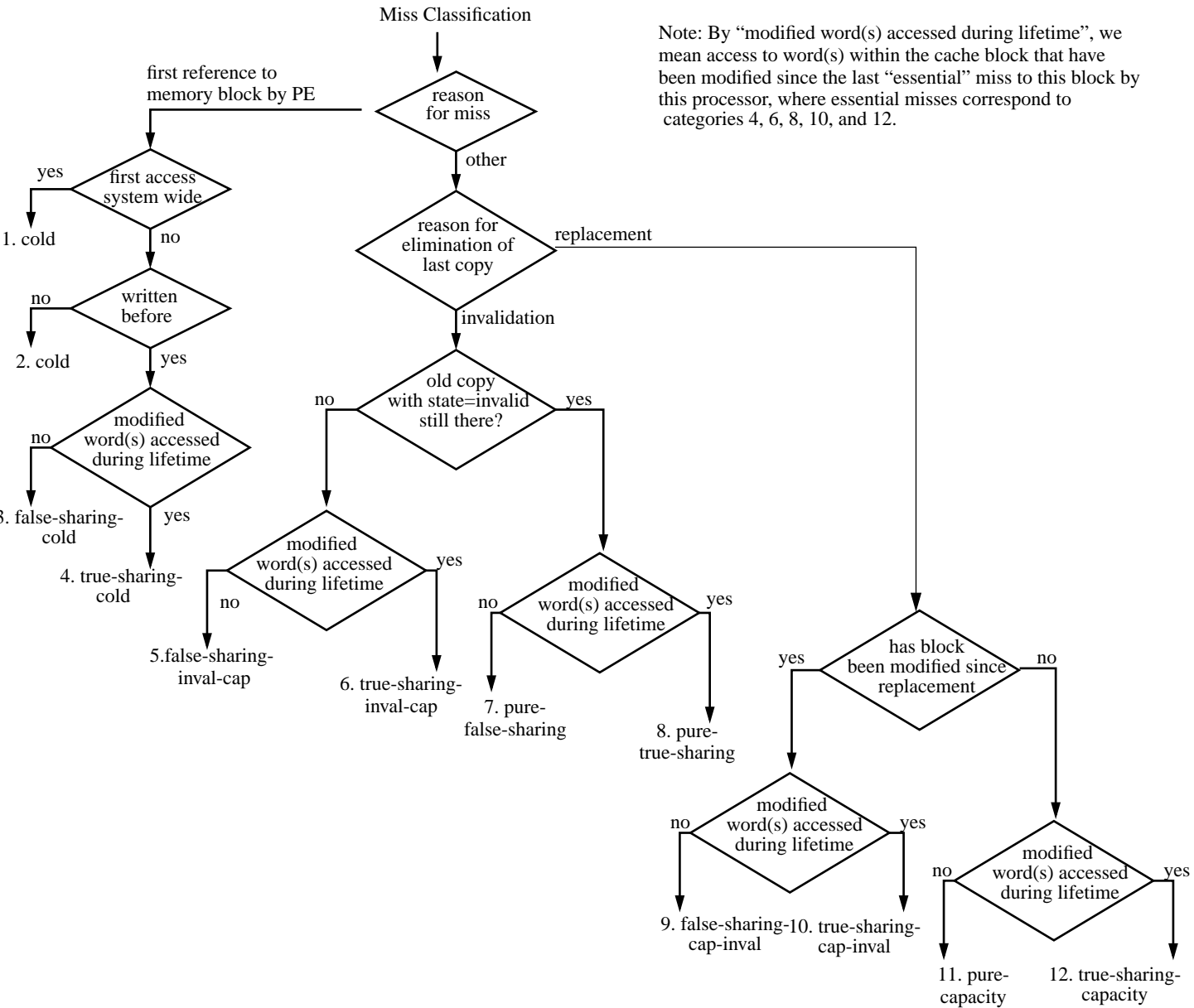
Note: By "modified word(s) accessed during lifetime", we mean access to word(s) within the cache block that have been modified since the last "essential" miss to this block by this processor, where essential misses correspond to categories 4, 6, 8, 10, and 12.

first reference to memory block by PE

reason for miss

other

first access system wide

yes

1. cold

no

written before

no

2. cold

yes

modified word(s) accessed during lifetime

no

3. false-sharing-cold

yes

4. true-sharing-cold

reason for elimination of last copy

replacement

invalidation

old copy with state=invalid still there?

no

yes

modified word(s) accessed during lifetime

yes

no

5.false-sharing-inval-cap

6. true-sharing-inval-cap

modified word(s) accessed during lifetime

no

yes

7. pure-false-sharing

8. pure-true-sharing

has block been modified since replacement

yes

no

modified word(s) accessed during lifetime

no

yes

9. false-sharing-cap-inval

10. true-sharing-cap-inval

modified word(s) accessed during lifetime

no

yes

11. pure-capacity

12. true-sharing-capacity

Figure 5-20 A classification of cache misses for shared-memory multiprocessors.

The four basic categories of cache misses are cold, capacity, true-sharing, and false-sharing misses. Many mixed categories arise because there may be multiple causes for the miss. For example, a block may be first replaced from processor A's cache, then be written to by processor B, and then be read back by processor A, making it a capacity-cum-invalidation false/true sharing miss.

[Artist: please change
the legend text as follows:
UPGMR --> Upgrade
FSMR--> False Sharing
TSMR --> True Sharing
CAPMR --> Capacity
COLDMR --> Cold]
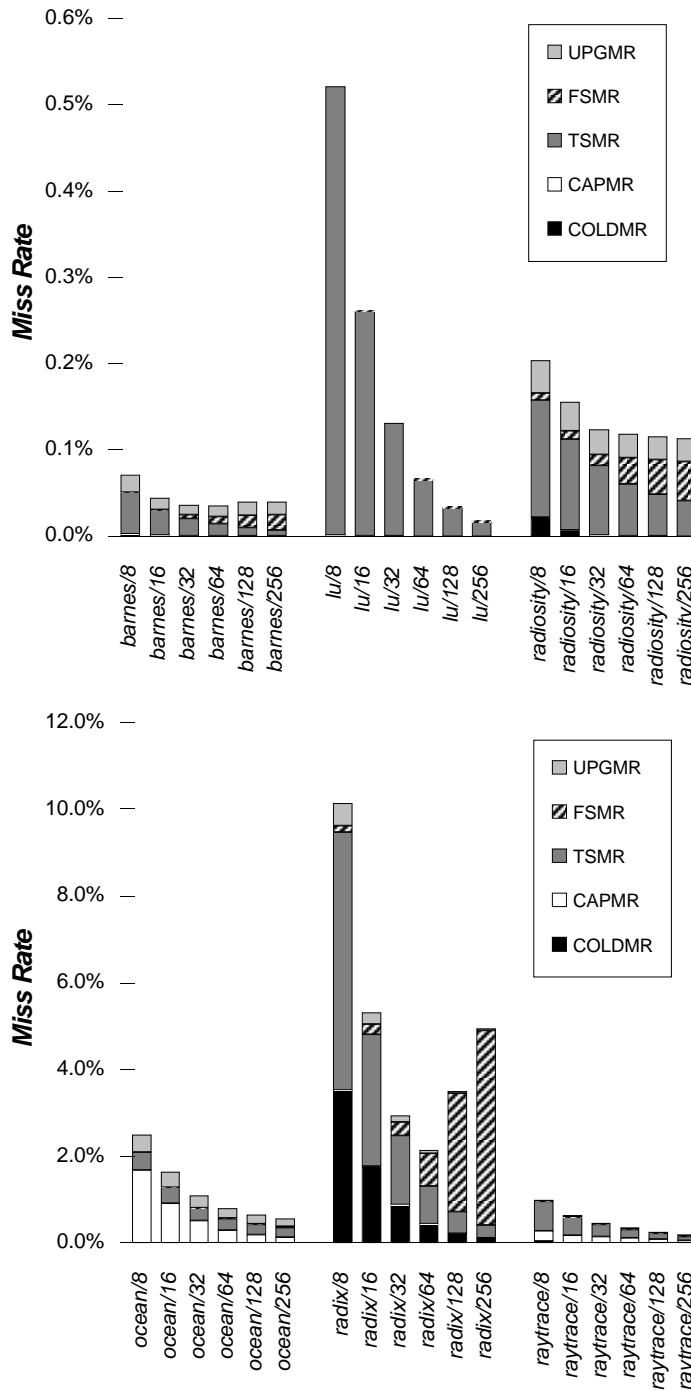
Figure 5-21 Breakdown of application miss rates as a function of cache block size for 1MB per-processor caches.

Conflict misses are included in capacity misses. The breakdown and behavior of misses varies greatly across applications, but there are some sommon trends. Cold misses and capacity misses tend to decrease quite quickly with block size due to spatial locality. True sharing misses. True sharing misses also tend to decrease, while false sharing misses increase. While the false sharing component is usually small for small block sizes, it sometimes remains small and sometimes increases very quickly.

For each individual application the miss characteristics change with block size much as we would expect, however, the overall characteristics differ widely across the sample programs. Cold, capacity and true sharing misses decrease with block size because the additional data brought in with each miss is accessed before the block is replaced, due to spatial locality. However, false sharing misses tend to increase with block size. In all cases, true sharing is a significant fraction of the misses, so even with ideal, infinite caches the miss rate and bus bandwidth will not go to zero. However, the size of the true sharing component varies significantly. Furthermore, some applications show substantial increase in false sharing with block size, while others show almost none. Below we investigate the properties of the applications that give rise to these differences observed at the machine level.

Relation to Application Structure

Multi-word cache blocks exploit spatial locality by prefetching data surrounding the accessed address. Of course, beyond a point larger cache blocks can hurt performance by: (i) prefetching unneeded data (ii) causing increased conflict misses, as the number of distinct blocks that can be stored in a finite-cache decreases with increasing block size; and (iii) causing increased false-sharing misses. Spatial locality in parallel programs tends to be lower than in sequential programs because when a memory block is brought into the cache some of the data therein will belong to another processor and will not be used by the processor performing the miss. As an extreme example, some parallel decompositions of scientific programs assign adjacent elements of an array to be handled by different processors to ensure good load balance, and in the process substantially decrease the spatial locality of the program.

The data in Figure 5-21 show that LU and Ocean have both good spatial locality and little false sharing. There is good spatial locality because the miss-rates drop proportionately to increases in cache block size and there are essentially no false sharing misses. This is in large part because these matrix codes use architecturally-aware data-structures. For example, a grid in Ocean is not represented as a single 2-D array (which can introduce substantial false-sharing), but as a 2-D array of blocks each of which is itself a 2-D array. Such structuring, by programmers or compilers, ensures that most accesses are unit-stride and over small blocks of data, and thus the nice behavior. As for scaling, the spatial locality for these applications is expected to remain good and false-sharing non-existent both as the problem size is increased and as the number of processors is increased. This should be true even for cache blocks larger than 256 bytes.

The graphics application Raytrace also shows negligible false sharing and somewhat poor spatial locality. The false sharing is small because the main data structure (collection of polygons constituting the scene) is read-only. The only read-write sharing happens on the image-plane data structure, but that is well controlled and thus only a small factor. This true-sharing miss rate reduces well with increasing cache block size. The reason for the poor spatial locality of capacity misses (although the overall magnitude is small) is that the access pattern to the collection of polygons is quite arbitrary, since the set of objects that a ray will bounce off is unpredictable. As for scaling, as problem size is increased (most likely in the form of more polygons) the primary effect is likely to be larger capacity miss rates; the spatial locality and false-sharing should not change. A larger number of processors is in most ways similar to having a smaller problem size, except that we may see slightly more false sharing in the image-plane data structure.

The Barnes-Hut and Radiosity applications show moderate spatial locality and false sharing. These applications employ complex data structures, including trees encoding spatial information

and arrays where records assigned to each processor are not contiguous in memory. For example, Barnes-Hut operates on particle records stored in an array. As the application proceeds and particles move in physical space, particle records get reassigned to different processors, with the result that after some time adjacent particles most likely belong to different processors. True sharing misses then seldom see good spatial locality because adjacent records within a cache block are often not touched. For the same reason, false sharing becomes a problem at large block sizes, as different processors write to records that are adjacent within a cache block. Another reason for the false-sharing misses is that the particle data structure (record) brings together (i) fields that are being modified by the owner of that particle (e.g., the current force on this particle), and (ii) fields that are read-only by other processors and that are not being modified in this phase (e.g., the current position of the particle). Since these two fields may fall in the same cache block for large block sizes, false-sharing results. It is possible to eliminate such false-sharing by splitting the particle data structure but that is not currently done, as the absolute magnitude of the miss-rate is small. As problem size is scaled and number of processors is scaled, the behavior of Barnes-Hut is not expected to change much. This is because the working set size changes very slowly (as log of number of particles), spatial locality is determined by the record size of one particle and thus remains the same, and finally because the sources of false sharing are not sensitive to number of processors. Radiosity is, unfortunately, a much more complex application whose behavior is difficult to reason about with larger data sets or more processors; the only option is to gather empirical data showing the growth trends.

The poorest sharing behavior is exhibited by Radix, which not only has a very high miss rate (due to cold and true sharing misses), but which gets significantly worse due to false sharing misses for block sizes of 128 bytes or more. The false-sharing in Radix arises as follows. Consider sorting 256K keys, using a radix of 1024, and 16 processors. On average, this results in 16 keys per radix per processor (64 bytes of data), which are then written to a contiguous portion of a global array at a random (for our purposes) starting point. Since 64 bytes is smaller than the 128 byte cache blocks, the high potential for false-sharing is clear. As the problem size is increased (e.g., to 4 million keys above), it is clear that we will see much less false sharing. The effect of increasing number of processors is exactly the opposite. Radix illustrates clearly that it is not sufficient to look at a given problem size, a given number of processors, and based on that draw conclusions of whether false-sharing or spatial locality are or are not a problem. It is very important to understand how the results are dependent on the particular parameters chosen in the experiment and how these parameters may vary in reality.

Data for the Multiprog workload for 1 Mbyte caches are shown in Figure 5-22. The data are shown separately for user-code, user-data, kernel-code, and kernel data. As one would expect, for code, there are only cold and capacity misses. Furthermore, we see that the spatial locality is quite low. Similarly, we see that the spatial locality in data references is quite low. This is true for the application data misses, because gcc (the main application causing misses in Multiprog) uses a large number of linked lists, which obviously do not offer good spatial locality. It is also somewhat interesting that we have an observable fraction of true-sharing misses, although we are running only sequential applications. They arise due to process migration; when a sequential process migrates from one processor to another and then references memory blocks that it had written while it was executing on the other processor, they appear as true sharing misses. We see that the kernel data misses, far outnumber the user data misses and have just as poor spatial locality. While the spatial locality in cold and capacity misses is quite reasonable, the true-sharing misses do not decrease at all for kernel data.

[Artist: same change to legend as in previous figure.]
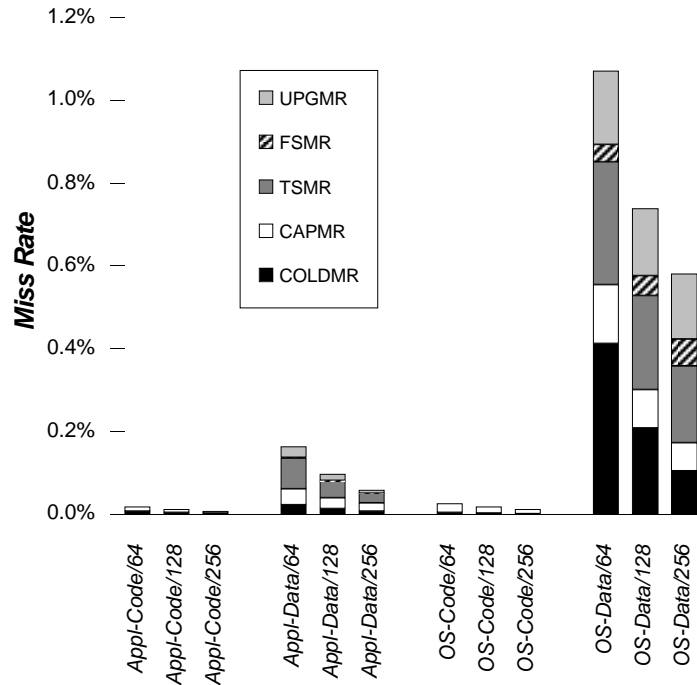
Figure 5-22 Breakdown of miss rates for Multiprog as a function of cache block size.

The results are for 1MB caches. Spatial locality for true sharing misses is much better for the applications than for the OS.

It is also interesting to look at the behavior of Ocean, Radix, and Raytrace for smaller 64 Kbyte caches. The miss-rate results are shown in Figure 5-23. As expected, the overall miss rates are higher and capacity misses have increased substantially. However, the spatial locality and false-sharing trends are not changed significantly as compared to the results for 1 Mbyte caches, mostly because these properties are fairly fundamental to data structure and algorithms used by a program and are not too sensitive to cache size. The key new interaction is for the Ocean application, where the capacity misses indicate somewhat poor spatial locality. The reason is that because of the small cache size data blocks are being thrown out of the cache due to interference before the processor has had a chance to reference all of the words in a cache block. Results for false sharing and spatial locality for other applications can be found in the literature [TLH94,JeE91].

Impact of Block Size on Bus Traffic

Let us briefly examine impact of cache-block size on bus traffic rather than miss rate. Note that while the number of misses and total traffic generated are clearly related, their impact on observed performance can be quite different. Misses have a cost that may contribute directly to performance, even though modern microprocessors try hard to hide the latency of misses by overlapping it with other activities. Traffic, on the other hand, affects performance only indirectly by causing contention and hence increasing the cost of other misses. For example, if an application program's misses are halved by increasing the cache block size but the bus traffic is doubled, this might be a reasonable trade-off if the application was originally using only 10% of the available bus and memory bandwidth. Increasing the bus/memory utilization to 20% is unlikely to

[Artist: same change to legend as in previous figure.]
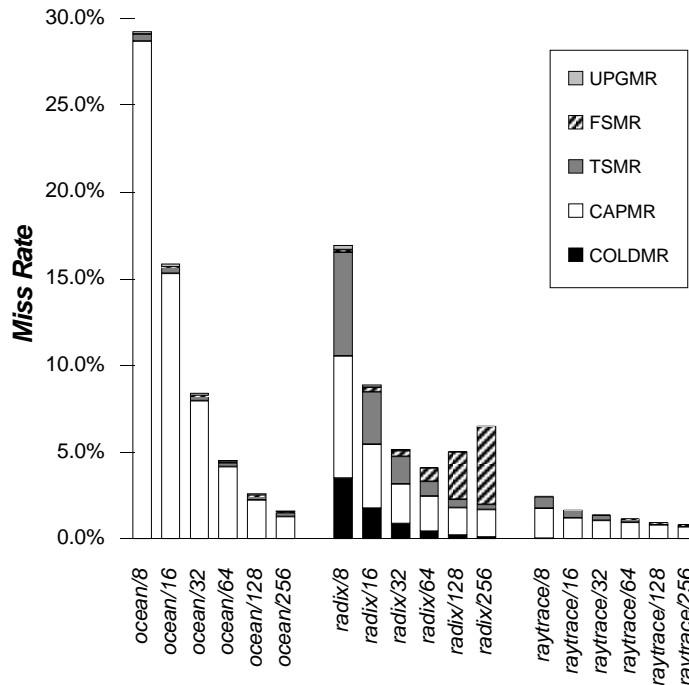
Figure 5-23 Breakdown of application miss-rates as a function of cache block size for 64 Kbyte caches.

Capacity misses are now a much larger fraction of the overall miss rate. Capacity miss rate decreases differently with block size for different applications.

increase the miss latencies significantly. However, if the application was originally using 75% of the bus/memory bandwidth, then increasing the block size is probably a bad idea under these circumstances

Figure 5-24 shows the total bus traffic in bytes/instruction or bytes/FLOP as block size is varied. There are three points to observe from this graph. Firstly, although overall miss rate decreases for most of the applications for the block sizes presented, traffic behaves quite differently. Only LU shows monotonically decreasing traffic for these block sizes. Most other applications see a doubling or tripling of traffic as block size becomes large. Secondly, given the above, the overall traffic requirements for the applications are still small for very large block sizes, with the exception of Radix. Radix's large bandwidth requirements (~650 Mbytes/sec per-processor for 128-byte cache blocks, assuming 200 MIPS processor) reflect its false sharing problems at large block sizes. Finally, the constant overhead for each bus transaction comprises a significant fraction of total traffic for small block sizes. Hence, although actual data traffic increases as we increase the block size, due to poor spatial locality, the total traffic is often minimized at 16-32 bytes. Figure 5-25 shows the data for Multiprog. While the bandwidth increase from 64 byte cache blocks to 128 byte blocks is small, the bandwidth requirements jump substantially at 256 byte cache blocks (primarily due to kernel data references). Finally, in Figure 5-26 we show traffic results for 64 Kbyte caches. Notice that for Ocean, even 64 and 128 byte cache blocks do not look so bad.
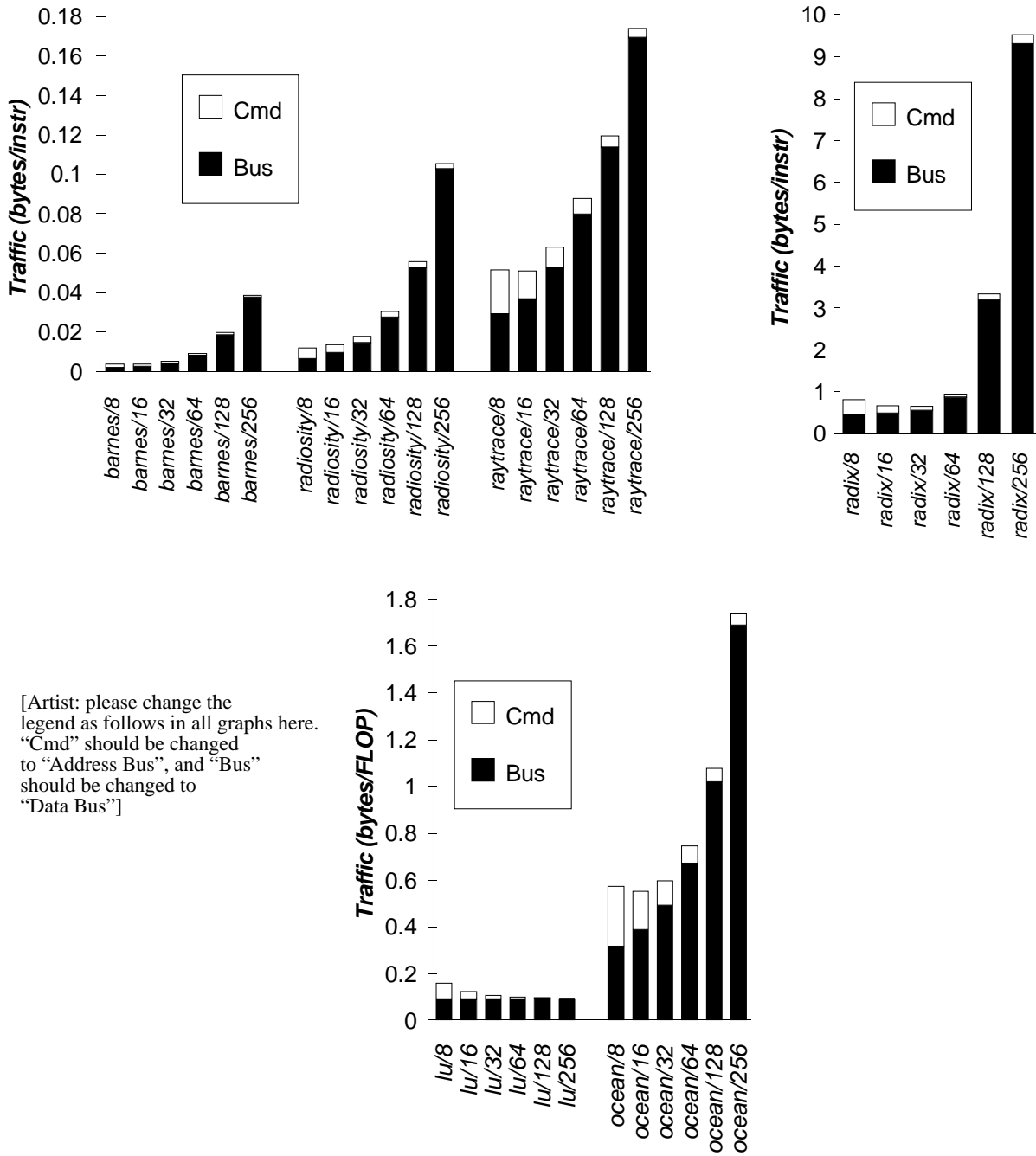
Figure 5-24 Traffic (in bytes/instr or bytes/flop) as a function of cache block size with 1 Mbyte per-processor caches.

Data traffic increases quite quickly with block size when communication misses dominate, except for applications like LU that have excellent spatial locality on all types of misses. Address and command bus traffic tends to decrease with block size since the miss rate and hence number of blocks transferred decreases.
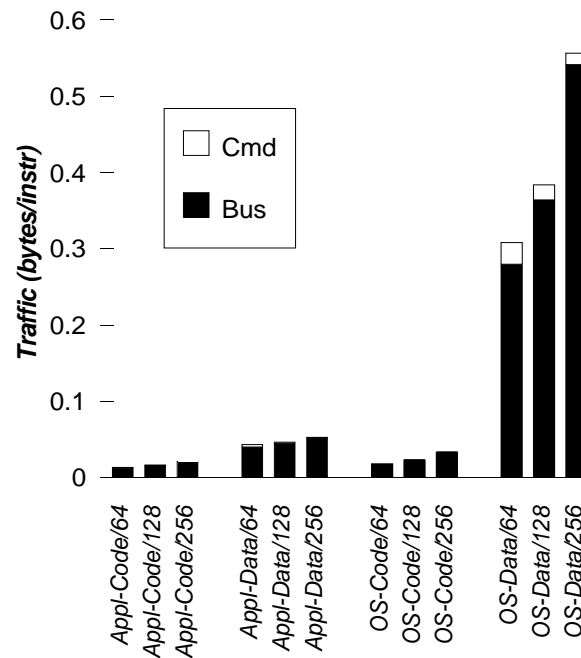
Figure 5-25 Traffic in bytes/instruction as a function of cache block size for Multiprog with 1Mbyte caches.

Traffic increasese quickly with block size for data references from the OS kernel.

Alleviating the Drawbacks of Large Cache Blocks

The trend towards larger cache block sizes is driven by the increasing gap between processor performance and memory access time. The larger block size amortizes the cost of the bus transaction and memory access time across a greater amount of data. The increasing density of processor and memory chips makes it possible to employ large first level and second level caches, so the prefetching obtained through a larger block size dominates the small increase in conflict misses. However, this trend potentially bodes poorly for multiprocessor designs, because false sharing becomes a larger problem. Fortunately there are hardware and software mechanisms that can be employed to counter the effects of large block size.

The software techniques to reduce false sharing are discussed later in the chapter. These essentially involve organizing data structures or work assignment so that data accessed by different processes is not at nearby addresses. One prime example is the use of higher dimensional arrays so blocks are contiguous. Some compiler techniques have also been developed to automate some of these techniques of laying out data to reduce false sharing [JeE91].

A natural hardware mechanism is the use of sub-blocks. Each cache block has a single address tag but distinct state bits for each of several sub-blocks. One subblock may be valid while others are invalid (or dirty). This technique is used in many machines to reduce the memory access time on a read miss, by resuming the processor when the accessed sub-block is present, or to reduce the amount of data that is copied back to memory on a replacement. Under false sharing, a write by one processor may invalidate the sub-block in another processors cache while leaving the other sub-blocks valid. Alternatively, small cache blocks can be used, but on a miss prefetch
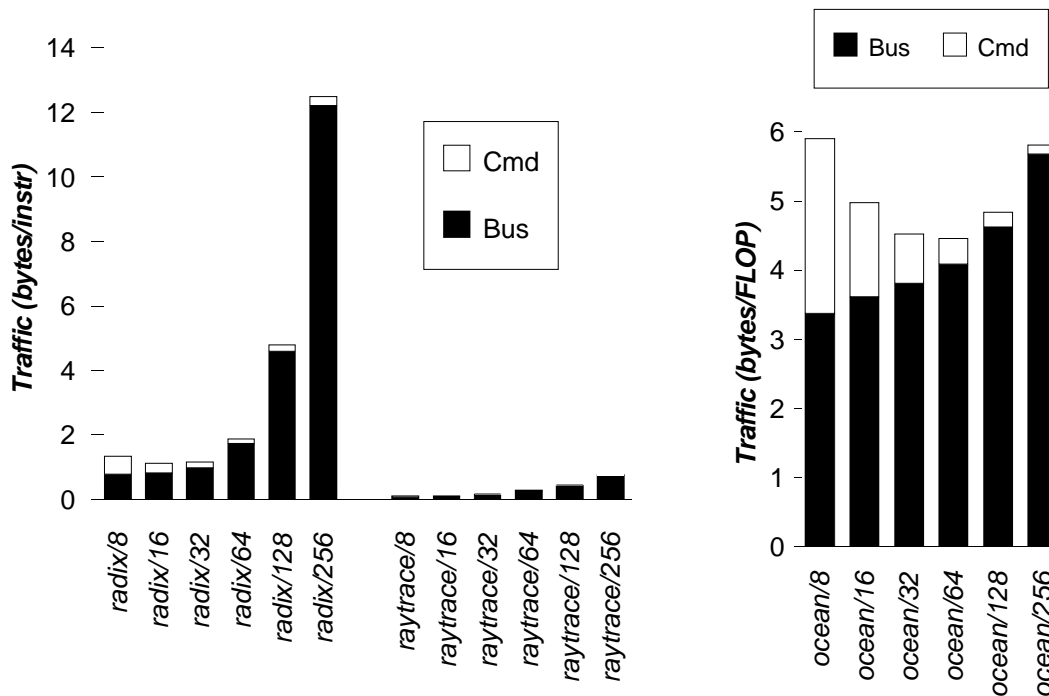
Figure 5-26 Traffic (bytes/instr or bytes/flop) as a function of cache block size with 64 Kbyte per-processor caches.

Traffic increases more slowly now for Ocean than with 1MB caches, since the capacity misses that now dominate exhibit excellent spatial locality (traversal of a process's assigned subgrid). However, traffic in Radix increases quickly once the threshold block size that causes false sharing is exceeded.

blocks beyond the accessed block. Proposals have also been made for caches with adjustable block size [DuL92].

A more subtle hardware technique is to delay generating invalidations until multiple writes have been performed. Delaying invalidations and performing them all at once reduces the occurrence of intervening read misses to false shared data. However, this sort of technique can change the memory consistency model in subtle ways, so further discussion is deferred until Chapter 6 where we consider weaker consistency models in the context of large scale machines. The other hardware technique that is important to consider is the use of update protocols, which push updates into other cached blocks, rather than invalidating those blocks. This option has been the subject of considerable study.

5.5.4 Update-based vs. Invalidation-based Protocols

A larger design issue is the question of whether writes should cause cached copies to be updated or invalidated. This has been the subject of considerable debate and various companies have taken different stands and, in fact, changed their position from one design to the next. The controversy on this topic arises because the relative performance of machines using an update-based versus an invalidation-based protocol depends strongly on the sharing patterns exhibited by the workload that is run and on the assumptions on cost of various underlying operations. Intuitively,

if the processors that were using the data before it was updated are likely to continue to use it in the future, updates should perform better than invalidates. However, if the processor holding the old data is never going to use it again, the updates are useless and just consume interconnect and controller resources. Invalidates would clean out the old copies and eliminate the apparent sharing. This latter 'pack rat' phenomenon is especially irritating under multiprogrammed use of an SMP, when sequential processes migrate from processor to processor so the apparent sharing is simply a shadow of the old process footprint. It is easy to construct cases where either scheme does substantially better than the other, as illustrated by the following example.

Example 5-12   Consider the following two program reference patterns:

Pattern-1: Repeat k times, processor-1 writes new value into variable V and processors 2 through N read the value of V. This represents a one-producer many-consumer scenario, that may arise for example when processors are accessing a highly-contended spin-lock.

Pattern-2: Repeat k times, processor-1 writes M-times to variable V, and then processor-2 reads the value of V. This represents a sharing pattern that may occur between pairs of processors, where the first accumulates a value into a variable, and then when the accumulation is complete it shares the value with the neighboring processor.

What is the relative cost of the two protocols in terms of the number of cache-misses and the bus traffic that will be generated by each scheme? To make the calculations concrete, assume that an invalidation/upgrade transaction takes 6 bytes (5 bytes for address, plus 1 byte command), an update takes 14 bytes (6 bytes for address and command, and 8 bytes of data), and that a regular cache-miss takes 70 bytes (6 bytes for address and command, plus 64 bytes of data corresponding to cache block size). Also assume that N=16, M=10, k=10, and that initially all caches are empty.

Answer   With an update scheme, on pattern 1 the first iteration on all N processors will incur a regular cache miss. In subsequent k-1 iterations, no more misses will occur and only one update per iteration will be generated. Thus, overall, we will see: misses = N = 16; traffic = N x RdMiss + (k-1). Update = 16 x 70 + 10 x 14 = 1260 bytes.

With an invalidate scheme, in the first iteration all N processors will incur a regular cache miss. In subsequent k-1 iterations, processor-1 will generate an upgrade, but all others will experience a read miss. Thus, overall, we will see: misses = N + (k-1) x (N-1) = 16 + 9 x 15 = 151; traffic = misses x RdMiss + (k-1). Upgrade = 151 x 70 + 9 x 6 = 10,624 bytes.

With an update scheme on pattern 2, in the first iteration there will be two regular cache misses, one for processor-1 and the other for processor-2. In subsequent k-1 iterations, no more misses will be generated, but M updates will be generated each iteration. Thus, overall, we will see: misses = 2; traffic = 2. RdMiss + M x (k-1). Update = 2 x 70 + 10 x 9 x 14 = 1400 bytes.

With an invalidate scheme, in the first iteration there will be two regular cache miss. In subsequent k-1 iterations, there will be one upgrade plus one regular miss each

iteration. Thus, overall, we will see: misses = 2 + (k-1) = 2 + 9 = 11; traffic = misses x RdMiss + (k-1) x Upgrade = 11 x 70 + 9 x 6 = 824 bytes.

The example above shows that for pattern-1 the update scheme is substantially superior, while for pattern-2 the invalidate scheme is substantially superior. This anecdotal data suggests that it might be possible design schemes that capture the advantages of both update and invalidate protocols. The success of such schemes will depend on the sharing patterns for real parallel programs and workloads. Let us briefly explore the design options and then employ the workload driven evaluation of Chapter 4.

Combining Update and Invalidation-based Protocols

One way to take advantage of both update and invalidate protocols is to support both in hardware and then to decide dynamically at a page granularity, whether coherence for any given page is to be maintained using an update or an invalidate protocol. The decision about choice of protocol to use can be indicated by making a system call. The main advantage of such schemes is that they are relatively easy to support; they utilize the TLB to indicate to the rest of the coherence subsystem which of the two protocols to use. The main disadvantage of such schemes is the substantial burden they put on the programmer. The decision task is also made difficult because of the coarse-granularity at which control is made available.

An alternative is to make the decision about choice of protocol at a cache-block granularity, by observing the sharing behavior at run time. Ideally, for each write, one would like to be able to peer into the future references that will be made to that location by all processors and then decide (in a manner similar to what we used above to compute misses and traffic) whether to invalidate other copies or to do an update. Since this information is obviously not available, and since there are substantial perturbations to an ideal model due to cache replacements and multi-word cache blocks, a more practical scheme is needed. So called "competitive" schemes change the protocol for a block between invalidate and update based on observed patterns at runtime. The key attribute of any such scheme is that if a wrong decision is made once for a cache block, the losses due to that wrong decision should be kept bounded and small [KMR+86]. For example, if a block is currently using update mode, it should not remain in that mode if one processor is continuously writing to it but none of the other processors are reading values from that block.

As an example, one class of schemes that has been proposed to bound the losses of update protocols works as follows [GSD95]. Starting with the base Dragon update protocol as described in Section 5.4.3, associate a count-down counter with each block. Whenever a given cache block is accessed by the local processor, the counter value for that block is reset to a threshold value, k. Every time an update is received for a block, the counter is decremented. If the counter goes to zero, the block is locally invalidated. The consequence of the local invalidation is that the next time an update is generated on the bus, it may find that no other cache has a valid (shared) copy, and in that case (as per the Dragon protocol) that block will switch to the modified (exclusive) state and will stop generating updates. If some other processor accesses that block, the block will again switch to shared state and the protocol will again start generating updates. A related approach implemented in the Sun SPARCcenter 2000 is to selectively invalidate with some probability, which is a parameter set when configuring the machine [Cat94]. Other mixed approaches may also be used. For example, one system uses an invalidation-based protocol for first-level caches and by default an update-based protocol for the second-level caches. However, if the L2 cache receives a second update for the block while the block in the L1 cache is still invalid, then

the block is invalidated in the L2 cache as well. When the block is thus invalidated in all other L2 caches, writes to the block are no longer placed on the bus.

Workload-driven Evaluation

To assess the tradeoffs among invalidate, update, and the mixed protocols just described, Figure 5-27 shows the miss rates, by category for four applications using 1 MBytes, 4-way set associative caches with a 64 byte block size. The mixed protocol used is the first one discussed in the previous paragraph. We see that for applications with significant capacity miss rates, this category increases with an update protocol. This make sense, because the protocol keeps data in processor caches that would have been removed by an invalidation protocol. For applications with significant true-sharing or false-sharing miss rates, these categories decrease with an invalidation protocol. After a write update, the other caches holding the blocks can access them without a miss. Overall, for these categories the update protocol appears to be advantageous and the mixed protocol falls in between. The category that is not shown in this figure is the upgrade and update operations for these protocols. This data is presented in Figure 5-28. Note that the scale of the graphs have changed because update operations are roughly four times more prevalent than misses. It is useful to separate these operations from other misses, because the way they are handled in the machine is likely to be different. Updates are a single word write, rather than a full cache block transfer. Because the data is being pushed from where it is being produced, it may arrive at the consumer before it is even needed, so the latency of these operations may be less critical than misses.

[ARTIST: same change to legend as before (fig 5-22, except please get rid of UPGMR altogether since it is not included in this figure.]
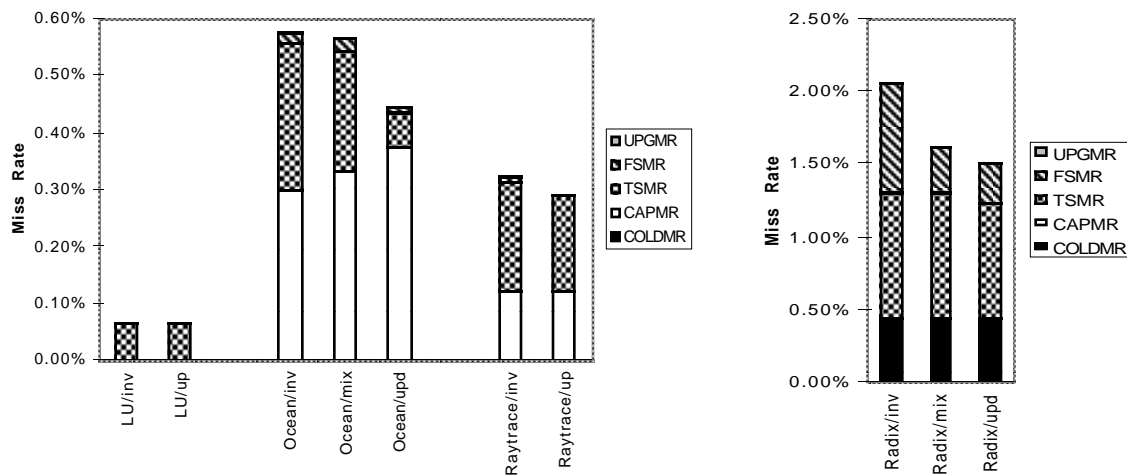
Figure 5-27 Miss rates and their decomposition for invalidate, update, and hybrid protocols.

1Mbyte caches, 64byte cache blocks, 4-way set-associativity, and threshold k=4 for hybrid protocol.

The traffic associated with updates is quite substantial. In large part this occurs where multiple writes are made to the same block. With the invalidate protocol, the first of these writes may

cause an invalidation, but the rest can simply accumulate in the block and be transferred in one bus transaction on a flush or a write-back. Sophisticated update schemes might attempt to delay the update to achieve a similar effect (by merging writes in the write buffer), or use other techniques to reduce traffic and improve performance [DaS95]. However, the increased bandwidth demand, the complexity of supporting updates, the trend toward larger cache blocks and the pack-rat phenomenon with sequential workloads underly the trend away from update-based protocols in the industry. We will see in Chapter 8 that update protocols also have some other prob-

[ARTIST: Please remove FSMR, TSMR, CAPMR and COLDMR from legend, and please change the only remaining one, (UPGMR) to Upgrade/Update].
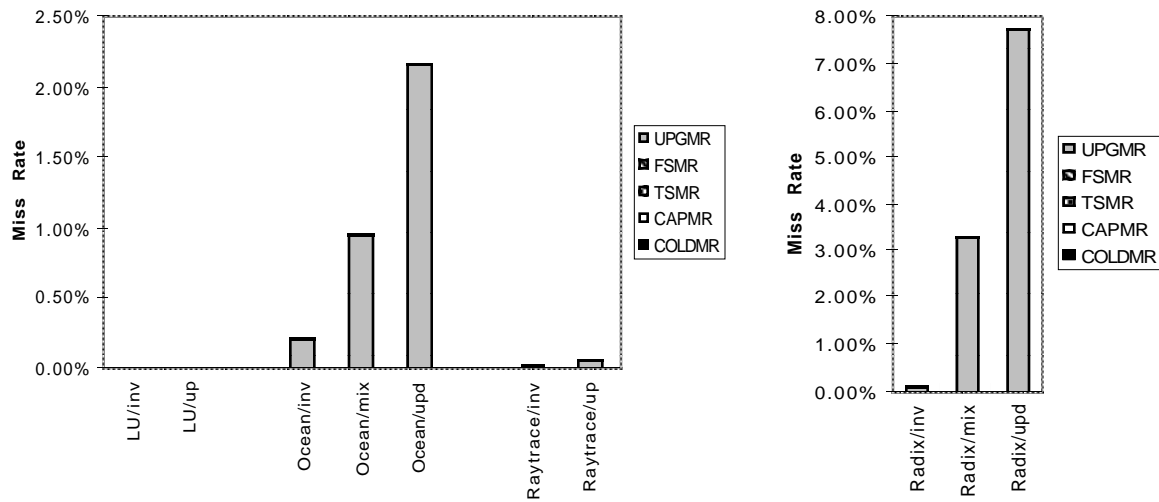
Figure 5-28 Upgrade and update rates for invalidate, update, and mixed protocols

1Mbyte, 64byte, 4-way, and k=4

lems for scalable cache-coherent architectures, making it less attractive for microprocessors to support them.

5.6 Synchronization

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations, mutual exclusion, point-to-point events and global events, and there has been considerable debate over the years on what hardware primitives should be provided in multiprocessor machines to support these synchronization operations. The conclusions have changed from time to time, with changes in technology and machine design style. Hardware support has the advantage of speed, but moving functionality to software has the advantage of flexibility and adaptability to different situations. The classic work of Dijkstra [Dij65] and Knuth [Knu66] shows that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization operations rely on some *atomic read-modify-write* machine primitive, in which the value of a memory loca-

tion is read, modified and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built using these primitives.

The history of instruction set design offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare-and-swap* instruction, to support synchronization in multiprogramming on uniprocessor or multiprocessor systems. The compare&swap compares the value in a memory location with the value in a specified register, and if they are equal swaps the value of the location with the value in a second register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic, so with the source and destination being memory operands much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high level language architecture have proposed that the user level synchronization operations, such a locks and barriers, should be supported at the machine level, not just the atomic read-modify-write primitives; i.e. the synchronization "algorithm" itself should be implemented in hardware. The issue became very active with the reduced instruction set debates, since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The SPARC approach was to provide atomic operations involving a register and a memory location, e.g., a simple swap (atomically swap the contents of the specified register and memory location) and a compare-and-swap, while MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described below, which allow higher level synchronization operations to be constructed without placing the burden of full read-modify-write instructions on the machine implementor. In essence, the pair of instructions can be used to implement atomic exchange (or higher-level operations) instead of a single instruction. This approach was later incorporated in the PowerPC and DEC Alpha architectures, and is now quite popular. Synchronization brings to light an unusually rich family of tradeoffs across the layers of communication architecture.

The focus of this section is how synchronization operations can be implemented on a bus-based cache-coherent multiprocessor. In particular, it describes the implementation of mutual exclusion through lock-unlock pairs, point-to-point event synchronization through flags, and global event synchronization through barriers. Not only is there a spectrum of high level operations and of low level primitives that can be supported by hardware, but the synchronization requirements of applications vary substantially. Let us begin by considering the components of a synchronization event. This will make it clear why supporting the high level mutual exclusion and event operations directly in hardware is difficult and is likely to make the implementation too rigid. Given that the hardware supports only the basic atomic state transitions, we can examine role of the user software and system software in synchronization operations, and then examine the hardware and software design tradeoffs in greater detail.

5.6.1 Components of a Synchronization Event

There are three major components of a given type of synchronization event:

an *acquire method*: a method by which a process tries to acquire the right to the synchronization (to enter the critical section or proceed past the event synchronization)

a *waiting algorithm*: a method by which a process waits for a synchronization to become available when it is not. For example, if a process tries to acquire a lock but the lock is not free (or proceed past an event but the event has not yet occurred), it must somehow wait until the lock becomes free.

a *release method*: a method for a process to enable other processes to proceed past a synchronization event; for example, an implementation of the UNLOCK operation, a method for the last processes arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

The choice of waiting algorithm is quite independent of the type of synchronization: What should a processor that has reached the acquire point do while it waits for the release to happen? There are two choices here: *busy-waiting* and *blocking*. Busy-waiting means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the process to proceed. Under blocking, the process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait. It will be awoken and made ready to run again when the release it was waiting for occurs. The tradeoffs between busy-waiting and blocking are clear. Blocking has higher overhead, since suspending and resuming a process involves the operating system, and suspending and resuming a thread involves the runtime system of a threads package, but it makes the processor available to other threads or processes with useful work to do. Busy-waiting avoids the cost of suspension, but consumes the processor and memory system bandwidth while waiting. Blocking is strictly more powerful than busy waiting, because if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.[1] Busy-waiting is likely to be better when the waiting period is short, whereas, blocking is likely to be a better choice if the waiting period is long and if there are other processes to run. Hybrid waiting methods can be used, in which the process busy-waits for a while in case the waiting period is short, and if the waiting period exceeds a certain threshold, blocks allowing other processes to run. The difficulty in implementing high level synchronization operations in hardware is not the acquire and release components, but the waiting algorithm. Thus, it makes sense to provide hardware support for the critical aspects of the acquire and release and allow the three components to be glued together in software. However, there remains a more subtle but very important hardware/software interaction in how the spinning operation in the busy-wait component is realized.

5.6.2 Role of User, System Software and Hardware

Who should be responsible for implementing the internals of high-level synchronization operations such as locks and barriers? Typically, a programmer wants to use locks, events, or even higher level operations and not have to worry about their internal implementation. The implementation is then left to the system, which must decide how much hardware support to provide and how much of the functionality to implement in software. Software synchronization algorithms using simple atomic exchange primitives have been developed which approach the speed of full hardware implementations, and the flexibility and hardware simplification they afford are

---

1. This problem of denying resources to the critical process or thread is one problem that is actually made simpler in with more processors. When the processes are timeshared on a single processor, strict busy-waiting without preemption is sure to be a problem. If each process or thread has its own processor, it is guaranteed not to be a problem. Realistic multiprogramming environments on a limited set of processors fall somewhere in between.

very attractive. As with other aspects of the system design, the utility of faster operations depends on the frequency of the use of those operations in the applications. So, once again, the best answer will be determined by a better understanding of application behavior.

Software implementations of synchronization constructs are usually included in system libraries. Many commercial systems thus provide subroutines or system calls that implement lock, unlock or barrier operations, and perhaps some types of other event synchronization. Good synchronization library design can be quite challenging. One potential complication is that the same type of synchronization (lock, barrier), and even the same synchronization variable, may be used at different times under very different runtime conditions. For example, a lock may be accessed with low-contention (a small number of processors, maybe only one, trying to acquire the lock at a time) or with high-contention (many processors trying to acquire the lock at the same time). The different scenarios impose different performance requirements. Under high-contention, most processes will spend time waiting and the key requirement of a lock algorithm is that it provide high lock-unlock bandwidth, whereas under low-contention the key goal is to provide low latency for lock acquisition. Since different algorithms may satisfy different requirements better, we must either find a good compromise algorithm or provide different algorithms for each type of synchronization among which a user can choose. If we are lucky, a flexible library can at runtime choose the best implementation for the situation at hand. Different synchronization algorithms may also rely on different basic hardware primitives, so some may be better suited to a particular machine than others. A second complication is that these multiprocessors are often used for multiprogrammed workloads where process scheduling and other resource interactions can change the synchronization behavior of the processes in a parallel program. A more sophisticated algorithm that addresses multiprogramming effects may provide better performance in practice than a simple algorithm that has lower latency and higher bandwidth in the dedicated case. All of these factors make synchronization a critical point of hardware/software interaction.

5.6.3 Mutual Exclusion

Mutual exclusion (lock/unlock) operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is little contention for the lock, but inefficient under high contention, whereas sophisticated algorithms that deal well with contention have a higher cost in the low contention case. After a brief discussion of hardware locks, the simplest algorithms for memory-based locks using atomic exchange instructions are described. Then, we discuss how the simplest algorithms can be implemented by using the special load locked and conditional store instruction pairs to synthesize atomic exchange, in place of atomic exchange instructions themselves, and what the performance tradeoffs are. Next, we discuss more sophisticated algorithms that can be built using either method of implementing atomic exchange.

Hardware Locks

Lock operations can be supported entirely in hardware, although this is not popular on modern bus-based machines. One option that was used one some older machines was to have a set of lock lines on the bus, each used for one lock at a time. The processor holding the lock asserts the line, and processors waiting for the lock wait for it to be released. A priority circuit determines which-gets the lock next when there are multiple requestors. However, this approach is quite inflexible since only a limited number of locks can be in use at a time and waiting algorithm is fixed (typically busy-wait with abort after timeout). Usually, these hardware locks were used only by the operating system for specific purposes, one of which was to implement a larger set of software

locks in memory, as discussed below. The Cray xMP provided an interesting variant of this approach. A set of registers were shared among the processors, including a fixed collection of lock registers[**XMP**]. Although the architecture made it possible to assign lock registers to user processes, with only a small set of such registers it was awkward to do so in a general purpose setting and, in practice, the lock registers were used primarily to implement higher level locks in memory.

Simple Lock Algorithms on Memory Locations

Consider a lock operation used to provide atomicity for a critical section of code. For the acquire method, a process trying to obtain a lock must check that the lock is free and if it is then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A simple way of thinking about the lock operation is that a process trying to obtain the lock should check if the variable is 0 and if so set it to 1 thus marking the lock busy; if the variable is 1 (lock is busy) then it should wait for the variable to turn to 0 using the waiting algorithm. An unlock operation should simply set the variable to 0 (the release method). Assembly-level instructions for this attempt at a lock and unlock are shown below (in our pseudo-assembly notation, the first operand always specifies the destination if there is one).

```
lock:     ld   register, location      /* copy location to register */
          cmp  location, #0            /* compare with 0 */
          bnz  lock                    /* if not 0, try again */
          st   location, #1            /* store 1 into location to mark it locked */
          ret                          /* return control to caller of lock */
```

and

```
unlock:   st location, #0             /* write 0 to location */
          ret                         /* return control to caller */
```

The problem with this lock, which is supposed to provide atomicity, is that it needs atomicity in its own implementation. To illustrate this, suppose that the lock variable was initially set to 0, and two processes P0 and P1 execute the above assembly code implementations of the lock operation. Process P0 reads the value of the lock variable as 0 and thinks it is free, so it enters the critical section. Its next step is to set the variable to 1 marking the lock as busy, but before it can do this process P1 reads the variable as 0, thinks the lock is free and enters the critical section too. We now have two processes simultaneously in the same critical section, which is exactly what the locks were meant to avoid. Putting the store of 1 into the location just after the load of the location would not help. The two-instruction sequence—reading (testing) the lock variable to check its state, and writing (setting) it to busy if it is free—is not atomic, and there is nothing to prevent these operations from different processes from being interleaved in time. What we need is a way to atomically test the value of a variable *and* set it to another value if the test succeeded (i.e. to atomically read and then conditionally modify a memory location), and to return whether the atomic sequence was executed successfully or not. One way to provide this atomicity for user processes is to place the lock routine in the operating system and access it through a system call, but this is expensive and leaves the question of how the locks are supported for the system itself. Another option is to utilize a hardware lock around the instruction sequence for the lock routine, but this also tends to be very slow compared to modern processors.

Hardware Atomic Exchange Primitives

An efficient, general purpose solution to the lock problem is to have an *atomic read-modify-write* instruction in the processor's instruction set. A typical approach is to have and atomic exchange instruction, in which a value at a location specified by the instruction is read into a register, and another value—that is either a function of the value read or not—is stored into the location, all in an atomic operation. There are many variants of this operation with varying degrees of flexibility in the nature of the value that can be stored. A simple example that works for mutual exclusion is an atomic *test&set* instruction. In this case, the value in the memory location is read into a specified register, and the constant 1 is stored into the location atomically if the value read is 0 (1 and 0 are typically used, though any other constants might be used in their place). Given such an instruction, with the mnemonic t&s, we can write a lock and unlock in pseudo-assembly language as follows:

```
lock:      t&s  register, location    /* copy location to reg, and if 0 set location to 1 */
           bnz  register, lock        /* compare old value returned with 0 */
                                      /* if not 0, i.e. lock already busy, try again */
           ret                        /* return control to caller of lock */
```

and

```
unlock:  st location, #0             /* write 0 to location */
         ret                         /* return control to caller */
```

The lock implementation keeps trying to acquire the lock using test&set instructions, until the test&set returns zero indicating that the lock was free when tested (in which case the test&set has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the location associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic test&set instruction.

More sophisticated variants of such atomic instructions exist, and as we will see are used by different software synchronization algorithms. One example is a *swap* instruction. Like a test&set, this reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location it writes whatever value was in the register to begin with. That is, it atomically exchanges or swaps the values in the memory location and the register. Clearly, we can implement a lock as before by replacing the test&set with a swap instruction as long as we ensure that the value in the register is 1 before the swap instruction is executed.

Another example is the family of so-called *fetch&op* instructions. A fetch&op instruction also specifies a location and a register. It atomically reads the value of the location into the register, and writes into the location the value obtained by applying to the current value of the location the operation specified by the fetch-and-op instruction. The simplest forms of fetch&op to implement are the fetch&increment and fetch&decrement instructions, which atomically read the current value of the location into the register and increment (or decrement) the value in the location by one. A fetch&add would take another operand which is a register or value to add into the previous value of the location. More complex primitive operations are possible. For example, the *compare&swap* operation takes two register operands plus a memory location; it compares the

value in the location with the contents of the first register operand, and if the two are equal it swaps the contents of the memory location with the contents of the second register.

Performance Issues

Figure 5-29 shows the performance of a simple test&set lock on the SGI Challenge.[1] Performance is measured for the following pseudocode executed repeatedly in a loop:

```
lock(L); critical-section(c); unlock(L);
```

where c is a delay parameter that determines the size of the critical section (which is only a delay, with no real work done). The benchmark is configured so that the same total number of locks are executed as the number of processors increases, reflecting a situation where there is a fixed number of tasks, independent of the number of processors. Performance is measured as the time per lock transfer, i.e., the cumulative time taken by all processes executing the benchmark divided by the number of times the lock is obtained. The uniprocessor time spent in the critical section itself (i.e. c times the number of successful locks executed) is subtracted from the total execution time, so that only the time for the lock transfers themselves (or any contention caused by the lock operations) is obtained. All measurements are in microseconds.

The upper curve in the figure shows the time per lock transfer with increasing number of processors when using the test&set lock with a very small critical section (ignore the curves with "back-off" in their labels for now). Ideally, we would like the time per lock acquisition to be independent of the number of processors competing for the lock, with only one uncontended bus transaction per lock transfer, as shown in the curve labelled ideal. However, the figure shows that performance clearly degrades with increasing number of processors. The problem with the test&set lock is that every attempt to check whether the lock is free to be acquired, whether successful or not, generates a write operation to the cache block that holds the lock variable (writing the value to 1); since this block is currently in the cache of some other processor (which wrote it last when doing its test&set), a bus transaction is generated by each write to invalidate the previous owner of the block. Thus, all processors put transactions on the bus repeatedly. The resulting contention slows down the lock considerably as the number of processors, and hence the frequency of test&sets and bus transactions, increases. The high degree of contention on the bus and the resulting timing dependence of obtaining locks causes the benchmark timing to vary sharply across numbers of processors used and even across executions. The results shown are for a particular, representative set of executions with different numbers of processors.

The major reason for the high traffic of the simple test&set lock above is the waiting method. A processor waits by repeatedly issuing test&set operations, and every one of these test&set operations includes a write in addition to a read. Thus, processors are consuming precious bus bandwidth even while waiting, and this bus contention even impedes the progress of the one process that is holding the lock (as it performs the work in its critical section and attempts to release the

---

1. In fact, the processor on the SGI Challenge, which is the machine for which synchronization performance is presented in this chapter, does not provide a test&set instruction. Rather, it uses alternative primitives that will be described later in this section. For these experiments, a mechanism whose behavior closely resembles that of test&set is synthesized from the available primitives. Results for real test&set based locks on older machines like the Sequent Symmetry can be found in the literature [GrT90, MCS87].
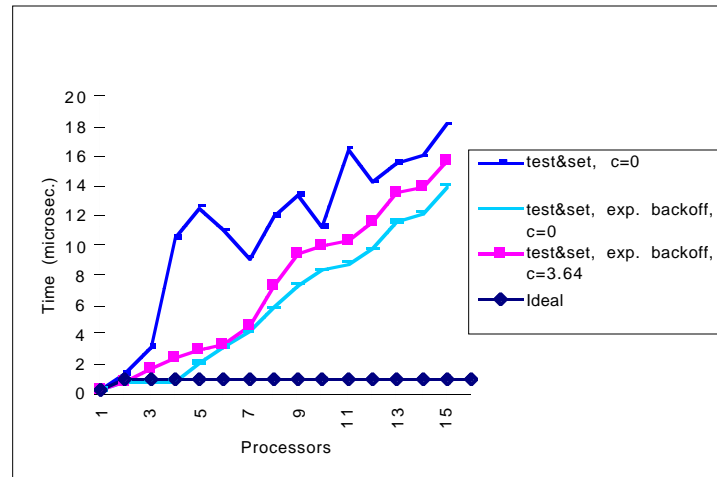
Figure 5-29 Performance of test&set locks with increasing number of competing processors on the SGI Challenge.

The Y axis is the time per lock-unlock pair, excluding the critical section of size c microseconds. The "exp. backoff" refers to exponential backoff, which will be discussed shortly. The irregular nature of the top curve is due to the timing-dependence of the contention effects caused. Note that since the processor on the SGI Challenge does not provide anatomic read-modify-write primitive but rather more sophisticated primitives discussed later in this section, the behavior of a test&set is simulated using those primitives for this experiment. Performance of locks that use test&set and test&set with backoff on older systems can be found in the literature [GrT90, MCS91].

lock).There are two simple things we can do to alleviate this traffic. First, we can reduce the frequency with which processes issue test&set instructions while waiting; second, we can have processes busy-wait only with read operations so they do not generate invalidations and misses until the lock is actually released. Let us examine these two possibilities, called the test&set lock with backoff and the test-and-test&set lock.

Test&set Lock with Backoff. The basic idea with backoff is to insert a delay after an unsuccessful attempt to acquire the lock. The delay between test&set attempts should not be too long, otherwise processors might remain idle even when the lock becomes free. But it should be long enough that traffic is substantially reduced. A natural question is whether the delay amount should be fixed or should vary. Experimental results have shown that good performance is obtained by having the delay vary "exponentially"; i.e. the delay after the first attempt is a small constant k, and then increases geometrically so that after the ith iteration it is $k*c^i$ where c is another constant. Such a lock is called a test&set lock with exponential backoff. Figure 5-29 also shows the performance for the test&set lock with backoff for two different sizes of the critical section and the starting value for backoff that appears to perform best. Performance improves, but still does not scale very well. Performance results using a real test&set instruction on older machines can be found in the literature [GrT90, MCS91]. See also Exercise 5.6, which discusses why the performance with a null critical section is worse than that with a non-zero critical section when backoff is used.

Test-and-test&set Lock. A more subtle change to the algorithm is have it use instructions that do not generate as much bus traffic while busy-waiting. Processes busy-wait by repeatedly *reading* with a standard load, not a test&set, the value of the lock variable until it turns from 1 (locked) to 0 (unlocked). On a cache-coherent machine, the reads can be performed in-cache by all processors, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes are invalidated, and the next

read of the variable by each process will generate a read miss. The waiting processes will then find that the lock has been made available, and will only then generate a test&set instruction to actually try to acquire the lock.

Before examining other lock algorithms and primitives, it is useful to articulate some performance goals for locks and to place the above locks along them. The goals include:

Low latency If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.

Low traffic Suppose many or all processors try to acquire a lock at the same time. They should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible. High traffic can slow down lock acquisitions due to contention, and can also slow down unrelated transactions that compete for the bus.

Scalability Related to the previous point, neither latency nor traffic should scale quickly with the number of processors used. Keep in mind that since the number of processors in a bus-based SMP is not likely to be large, it is not asymptotic scalability that is important.

Low storage cost The information needed for a lock should be small and should not scale quickly with the number of processors.

Fairness Ideally, processors should acquire a lock in the same order as their requests are issued. At least, starvation or substantial unfairness should be avoided.

Consider the simple atomic exchange or test&set lock. It is very low-latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, as discussed earlier it can generate a lot of bus traffic and contention if many processors compete for the lock. The scalability of the lock is poor with the number of competing processors. The storage cost is low (a single variable suffices) and does not scale with the number of processors. The lock makes no attempt to be fair, and an unlucky processor can be starved out. The test&set lock with backoff has the same uncontended latency as the simple test&set lock, generates less traffic and is more scalable, takes no more storage, and is no more fair. The test-and-test&set lock has slightly higher uncontended overhead than the simple test&set lock (it does a read in addition to a test&set even when there is no competition), but generates much less bus traffic and is more scalable. It too requires negligible storage and is not fair. Exercise 5.6 asks you to count the number of bus transactions and the time required for each type of lock.

Since a test&set operation and hence a bus transaction is only issued when a processor is notified that the lock is ready, and thereafter if it fails it spins on a cache block, there is no need for backoff in the test-and-test&set lock. However, the lock does have the problem that all processes rush out and perform both their read misses and their test&set instructions at about the same time when the lock is released. Each of these test&set instructions generates invalidations and subsequent misses, resulting in $O(p^2)$ bus traffic for p processors to acquire the lock once each. A random delay before issuing the test&set could help to stagger at least the test&set instructions, but it would increase the latency to acquire the lock in the uncontended case.

Improved Hardware Primitives: Load-locked, Store-conditional

Several microprocessors provide a pair of instructions called load locked and store conditional to implement atomic operations, instead of a atomic read-modify-write instructions like test&set. Let us see how these primitives can be used to implement simple lock algorithms and improve

performance over a test-and-test&set lock. Then, we will examine sophisticated lock algorithms that tend to use more sophisticated atomic exchange operations, which can be implemented either as atomic instructions or with load locked and store conditional.

In addition to spinning with reads rather than read-modify-writes, which test-and-test&set accomplishes, it would be nice to implement read-modify-write operations in such a way that failed attempts to complete the read-modify-write do not generate invalidations. It would also be nice to have a single primitive that allows us to implement a range of atomic read-modify-write operations—such as test&set, fetch&op, compare&swap—rather than implement each with a separate instruction. Using a pair of special instructions rather than a single instruction to implement atomic access to a variable, let's call it a synchronization variable, is one way to achieve both goals. The first instruction is commonly called *load-locked* or *load-linked* (LL). It loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register; i.e. the modify part of a read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional* (SC). It writes the register back to the memory location (the synchronization variable) *if and only if* no other processor has written *to that location* since this processor completed its LL. Thus, if the SC succeeds, it means that the LL-SC pair has read, perhaps modified in between, and written back the variable atomically. If the SC detects that an intervening write has occurred to the variable, it fails and does not write the value back (or generate any invalidations). This means that the atomic operation on the variable has failed and must be retried starting from the LL. Success or failure of the SC is indicated by the condition codes or a return value. How the LL and SC are actually implemented will be discussed later; for now we are concerned with their semantics and performance.

Using LL-SC to implement atomic operations, lock and unlock subroutines can be written as follows, where reg1 is the register into which the current value of the memory location is loaded, and reg2 holds the value to be stored in the memory location by this atomic exchange (reg2 could simply be 1 for a lock attempt, as in a test&set). Many processors may perform the LL at the same time, but only the first one that manages to put its store conditional on the bus will succeed in its SC. This processor will have succeeded in acquiring the lock, while the others will have failed and will have to retry the LL-SC.

```
lock:    ll   reg1, location       /* load-linked the location to reg1 */
         bnz  reg1, lock           /* if location was locked (nonzero), try again*/
         sc   location, reg2       /* store reg2 conditionally into location*/
         beqz lock                 /* if SC failed, start again*/
         ret                       /* return control to caller of lock */
```

and

```
unlock:  st location, #0           /* write 0 to location */
         ret                       /* return control to caller */
```

If the location is 1 (nonzero) when a process does its load linked, it will load 1 into reg1 and will retry the lock starting from the LL without even attempt the store conditional.

It is worth noting that the LL itself is not a lock, and the SC itself is not an unlock. For one thing, the completion of the LL itself does not guarantee obtaining exclusive access; in fact LL and SC are used together to implement a lock operation as shown above. For another, even a successful LL-SC pair does not guarantee that the instructions between them (if any) are executed atomi-

cally with respect to those instructions on other processors, so in fact those instructions do not constitute a critical section. All that a successful LL-SC guarantees is that no conflicting writes *to the synchronization variable* accessed by the LL and SC themselves intervened between the LL and SC. In fact, since the instructions between the LL and SC are executed but should not be visible if the SC fails, it is important that they do not modify any important state. Typically, they only manipulate the register holding the synchronization variable—for example to perform the op part of a fetch&op—and do not modify any other program variables (modification of the register is okay since the register will be re-loaded anyway by the LL in the next attempt). Microprocessor vendors that support LL-SC explicitly encourage software writers to follow this guideline, and in fact often provide guidelines on what instructions are possible to insert with guarantee of correctness given their implementations of LL-SC. The number of instructions between the LL and SC should also be kept small to reduce the probability of the SC failing. On the other hand, the LL and SC can be used directly to implement certain useful operations on shared data structures. For example, if the desired function is a shared counter, it makes much more sense to implement it as the natural sequence (LL, register op, SC, test) than to build a lock and unlock around the counter update.

Unlike the simple test&set, the spin-lock built with LL-SC does not generate invalidations if either the load-linked indicates that the lock is currently held or if the SC fails. However, when the lock is released, the processors spinning in a tight loop of load-locked operations will miss on the location and rush out to the bus with read transactions. After this, only a single invalidation will be generated for a given lock acquisition, by the processor whose SC succeeds, but this will again invalidate all caches. Traffic is reduced greatly from even the test-and-test&set case, down from $O(p^2)$ to $O(p)$ per lock acquisition, but still scales quickly with the number of processors. Since spinning on a locked location is done through reads (load-locked operations) there is no analog of a test-and-test&set to further improve its performance. However, backoff can be used between the LL and SC to further reduce bursty traffic.

The simple LL-SC lock is also low in latency and storage, but it is not a fair lock and it does not reduce traffic to a minimum. More advanced lock algorithms can be used that both provide fairness and reduce traffic. They can be built using both atomic read-modify-write instructions or LL-SC, though of course the traffic advantages are different in the two cases. Let us consider two of these algorithms.

Advanced Lock Algorithms

Especially when using a test&set to implement locks, it is desirable to have only one process attempt to obtain the lock when it is released (rather than have them all rush out to do a test&set and issue invalidations as in all the above cases). It is even more desirable to have only one process even incur a read miss when a lock is released. The ticket lock accomplishes the first purpose, while the array-based lock accomplishes both goals but at a little cost in space. Both locks are fair, and grant the lock to processors in FIFO order.

Ticket Lock. The ticket lock operates just like the ticket system in the sandwich line at a grocery store, or in the teller line at a bank. Every process wanting to acquire the lock takes a number, and busy-waits on a global *now-serving* number—like the number on the LED display that we watch intently in the sandwich line—until this *now-serving* number equals the number it obtained. To release the lock, a process simply increments the *now-serving* number. The atomic primitive needed is a fetch&increment, which a process uses to obtain its ticket number from a shared counter. It may be implemented as an atomic instruction or using LL-SC. No test&set is needed

to actually obtain the lock upon a release, since only the unique process that has its ticket number equal *now-serving* attempts to enter the critical section when it sees the release. Thus, the atomic primitive is used when a process first reaches the lock operation, not in response to a release. The acquire method is the fetch&increment, the waiting algorithm is busy-waiting for *now-serving* to equal the ticket number, and the release method is to increment *now-serving*. This lock has uncontended overhead about equal to the test-and-test&set lock, but generates much less traffic. Although every process does a fetch and increment when it first arrives at the lock (presumably not at the same time), the simultaneous test&set attempts upon a release of the lock are eliminated, which tend to be a lot more heavily contended. The ticket lock also requires constant and small storage, and is fair since processes obtain the lock in the order of their fetch&increment operations. However, like the simple LL-SC lock it still has a traffic problem. The reason is that all processes spin on the same variable (now-serving). When that variable is written at a release, all processors' cached copies are invalidated and they all incur a read miss. (The simple LL-SC lock was somewhat worse in this respect, since in that case another invalidation and set of read misses occurred when a processor succeeded in its SC.) One way to reduce this bursty traffic is to introduce a form of backoff. We do not want to use exponential backoff because we do not want all processors to be backing off when the lock is released so none tries to acquire it for a while. A promising technique is to have each processor backoff from trying to read the now-serving counter by an amount proportional to when it expects its turn to actually come; i.e. an amount proportional to the difference in its ticket number and the *now-serving* counter it last read. Alternatively, the array-based lock eliminates this extra read traffic upon a release completely, by having every process spin on a distinct location.

Array-based Lock. The idea here is to use a fetch&increment to obtain not a value but a unique location to busy-wait on. If there are p processes that might possibly compete for a lock, then the lock contains an array of p locations that processes can spin on, ideally each on a separate memory block to avoid false-sharing. The acquire method then uses a fetch&increment operation to obtain the next available location in this array (with wraparound) to spin on, the waiting method spins on this location, and the release method writes a value denoting "unlocked" to the next location in the array after the one that the releasing processor was itself spinning on. Only the processor that was spinning on that location has its cache block invalidated, and its consequent read miss tells it that it has obtained the lock. As in the ticket lock, no test&set is needed after the miss since only one process is notified when the lock is released. This lock is clearly also FIFO and hence fair. It's uncontended latency is likely to be similar to that of the test-and-test&set lock (a fetch&increment followed by a read of the assigned array location), and it is more scalable than the ticket lock since only one process incurs the read miss. It's only drawback for a bus-based machine is that it uses $O(p)$ space rather than $O(1)$, but with both p and the proportionality constant being small this is usually not a very significant drawback. It has a potential drawback for distributed memory machines, but we shall discuss this and lock algorithms that overcome this drawback in Chapter 7.

Performance

Let us briefly examine the performance of the different locks on the SGI Challenge, as shown in Figure 5-30. All locks are implemented using LL-SC, since the Challenge provides only these and not atomic instructions. The test&set locks are implemented by simulating a test&set using LL-SC, just as they were in Figure 5-29, and are shown as leaping off the graph for reference[1]. In particular, every time an SC fails a write is performed to another variable on the same cache block, causing invalidations as a test&set would. Results are shown for a somewhat more param-

eterized version of the earlier code for test&set locks, in which a process is allowed to insert a delay between its release of the lock and its next attempt to acquire it. That is, the code is a loop over the following body:

```
lock(L); critical_section(c); unlock(L); delay(d);
```

Let us consider three cases—(i) c=0, d=0, (ii) c=3.64 μs, d=0, and (iii) c=3.64 μs, d=1.29 μs— called *null*, *critical-section*, and *delay*, respectively. The delays c and d are inserted in the code as round numbers of processor cycles, which translates to these microsecond numbers. Recall that in all cases c and d (multiplied by the number of lock acquisitions by each processor) are subtracted out of the total time, which is supposed to measure the total time taken for a certain number of lock acquisitions and releases only (see also Exercise 5.6).

(a) null (c=0, d=0) (b) critical-section (c=3.64 μs, d=0) (c) delay (c=3.64 μs, d=1.29 μs)
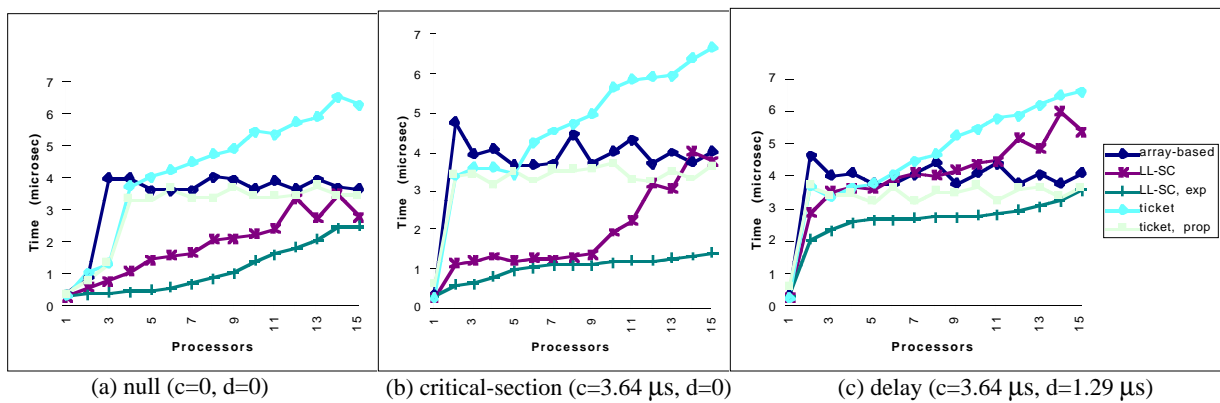
Figure 5-30 Performance of locks on the SGI Challenge, for three different scenarios.

Consider the null critical section case. The first observation, comparing with Figure 5-29, is that all the subsequent locks we have discussed are indeed better than the test&set locks as expected. The second observation is that the simple LL-SC locks actually perform better than the more sophisticated ticket lock and array-based lock. For these locks, that don't encounter so much contention as the test&set lock, with reasonably high-bandwidth busses the performance of a lock is largely determined by the number of bus transactions between a release and a successful acquire. The reason that the LL-SC locks perform so well, particularly at lower processor counts, is that they are not fair, and the unfairness is exploited by architectural interactions! In particular, when a processor that does a write to release a lock follows it immediately with the read (LL) for its next acquire, it's read and SC are likely to succeed in its cache before another processor can read the block across the bus. (The bias on the Challenge is actually more severe, since the releasing processor can satisfy its next read from its write buffer even before the corresponding read-exclusive gets out on the bus.) Lock transfer is very quick, and performance is good. As the number of processors increases, the likelihood of self-transfers decreases and bus traffic due to invalidations and read misses increases, so the time per lock transfer increases. Exponential backoff helps reduce the burstiness of traffic and hence slows the rate of scaling.

---

1. This method of simulating test&set with LL-SC may lead to somewhat worse performance than a true test&set primitive, but it conveys the trend.

At the other extreme (c=3.64, d=1.29), we see the LL-SC lock doing not quite so well, even at low processor counts. This is because a processor waits after its release before trying to acquire the lock again, making it much more likely that some other waiting processor will acquire the lock before it. Self-transfers are unlikely, so lock transfers are slower even at two processors. It is interesting that performance is particularly worse for the backoff case at small processor counts when the delay d between unlock and lock is non-zero. This is because with only a few processors, it is quite likely that while a processor that just released the lock is waiting for d to expire before doing its next acquire, the other processors are in a backoff period and not even trying to acquire the lock. Backoff must be used carefully for it to be successful.

Consider the other locks. These are fair, so every lock transfer is to a different processor and involves bus transactions in the critical path of the transfer. Hence they all start off with a jump to about 3 bus transactions in the critical path per lock transfer even when two processors are used. Actual differences in time are due to what exactly the bus transactions are and how much of their latency can be hidden from the processor. The ticket lock without backoff scales relatively poorly: With all processors trying to read the now-serving counter, the expected number of bus transactions between the release and the read by the correct processor is *p/2*, leading to the observed linear degradation in lock transfer critical path. With successful proportional backoff, it is likely that the correct processor will be the one to issue the read first after a release, so the time per transfer does not scale with *p*. The array-based lock also has a similar property, since only the correct processor issues a read, so its performance also does not degrade with more processors.

The results illustrate the importance of architectural interactions in determining the performance of locks, and that simple LL-SC locks perform quite well on busses that have high enough bandwidth (and realistic numbers of processors for busses). Performance for the unfair LL-SC lock scales to become as bad as or a little worse than for the more sophisticated locks beyond 16 processors, due to the higher traffic, but not by much because bus bandwidth is quite high. When exponential backoff is used to reduce traffic, the simple LL-SC lock delivers the best average lock transfer time in all cases. The results also illustrate the difficulty and the importance of sound experimental methodology in evaluating synchronization algorithms. Null critical sections display some interesting effects, but meaningful comparison depend on what the synchronization patterns look like in practice in real applications. An experiment to use LL-SC but guarantee round-robin acquisition among processors (fairness) by using an additional variable showed performance very similar to that of the ticket lock, confirming that unfairness and self-transfers are indeed the reason for the better performance at low processor counts.

Lock-free, Non-blocking, and Wait-free Synchronization

An additional set of performance concerns involving synchronization arise when we consider that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time or, even if we have the machine to ourselves, background daemons run periodically, processes take page faults, I/O interrupts occur, and the process scheduler makes scheduling decisions with limited information on the application requirements. These events can cause the rate at which processes make progress to vary considerably. One important question is how the program as a whole slows down when one process is slowed. With traditional locks the problem can be serious because if a process holding a lock stops or slows while in its critical section, all other processes may have to wait. This problem has received a good deal of attention in work on operating system schedulers and in some cases attempts are made to avoid preempting a process that is holding a lock. There is another line of research that takes the view

that lock-based operations are not very robust and should be avoided. If a process dies while holding a lock, other processes hang. It can be observed that many of the lock/unlock operations are used to support operations on a data structure or object that is shared by several processes, for example to update a shared counter or manipulate a shared queue. These higher level operation can be implemented directly using atomic primitives without actually using locks.

A shared data structure is *lock-free* if its operations do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in finite amount of time, even if other processes halt, the data structure is *non-blocking*. If the data structure operations can guantee that every (non-faulty) process will complete its operation in a finite amount of time, then the data structure is *wait-free*.[Her93]. There is a body of literature that investigates the theory and practice of such data structures, including requirements on the basic atomic primitives[Her88], general purpose techniques for translating sequential operations to non-blocking concurrent operations[Her93], specific useful lock-free data structures [Val95,MiSc96], operating system implementations [MaPu91,GrCh96] and proposals for architectural support[HeMo93]. The basic approach is to implement updates to a shared object by reading a portion of the object to make a copy, updating the copy, and then performing an operation to commit the change only if no conflicting updates have been made. As a simple example, consider a shared counter. The counter is read into a register, a value is added to the register copy and the result put in a second register, then a compare-and-swap is performed to update the shared counter only if its value is the same as the copy. For more sophisticated data structures a linked structure is used and typically the new element is linked into the shared list if the insert is still valid. These techniques serve to limit the window in which the shared data structure is in an inconsistent state, so they improve the robustness, although it can be difficult to make them efficient.

Having discussed the options for mutual exclusion on bus-based machines, let us move on to point-to-point and then barrier event synchronization.

5.6.4 Point-to-point Event Synchronization

Point-to-point synchronization within a parallel program is often implemented using busy-waiting on ordinary variables as flags. If we want to use blocking instead of busy-waiting, we can use semaphores just as they are used in concurrent programming and operating systems [TaW97].

Software Algorithms

Flags are control variables, typically used to communicate the occurrence of a synchronization event, rather than to transfer values. If two processes have a producer-consumer relationship on the shared variable *a*, then a flag can be used to manage the synchronization as shown below:

P1	P2
a = f(x); /* set a */	while (flag is 0) do nothing;
flag = 1;	b = g(a); /* use a */

If we know that the variable *a* is initialized to a certain value, say 0, which will be changed to a new value we are interested in by this production event, then we can use *a* itself as the synchronization flag, as follows:

	P1			P2	

a = f(x);   /* set a */   while (a is 0) do nothing;

                                                  b = g(a);   /* use a */

This eliminates the need for a separate flag variable, and saves the write to and read of that variable.

Hardware Support: Full-empty Bits

The last idea above has been extended in some research machines—although mostly machines with physically distributed memory—to provide hardware support for fine-grained producer-consumer synchronization. A bit, called a full-empty bit, is associated with every word in memory. This bit is set when the word is "full" with newly produced data (i.e. on a write), and unset when the word is "emptied" by a processor consuming those data (i.e. on a read). Word-level producer-consumer synchronization is then accomplished as follows. When the producer process wants to write the location it does so only if the full-empty bit is set to empty, and then leaves the bit set to full. The consumer reads the location only if the bit is full, and then sets it to empty. Hardware preserves the atomicity of the read or write with the manipulation of the full-empty bit. Given full-empty bits, our example above can be written without the spin loop as:

                   P1   P2

a = f(x);   /* set a */   b = g(a);   /* use a */

Full-empty bits raise concerns about flexibility. For example, they do not lend themselves easily to single-producer multiple-consumer synchronization, or to the case where a producer updates a value multiple times before a consumer consumes it. Also, should all reads and writes use full-empty bits or only those that are compiled down to special instructions? The latter requires support in the language and compiler, but the former is too restrictive in imposing synchronization on all accesses to a location (for example, it does not allow asynchronous relaxation in iterative equation solvers, see Chapter 2). For these reasons and the hardware cost, full-empty bits have not found favor in most commercial machines.

Interrupts

Another important kind of event is the interrupt conveyed from an I/O device needing attention to a processor. In a uniprocessor machine there is no question where the interrupt should go, but in an SMP any processor can potentially take the interrupt. In addition, there are times when one processor may need to issue an interrupt to another. In early SMP designs special hardware was provided to monitor the priority of the process on each processor and deliver the I/O interrupt to the processor running at lowest priority. Such measures proved to be of small value and most modern machines use simple arbitration strategies. In addition, there is usually a memory mapped interrupt control region, so at kernel level any processor can interrupt any other by writing the interrupt information at the associated address.

5.6.5 Global (Barrier) Event Synchronization

Software Algorithms

Software algorithms for barriers are typically implemented using locks, shared counters and flags. Let us begin with a simple barrier among *p* processes, which is called a centralized barrier since it uses only a single lock, a single counter and a single flag.

Centralized Barrier

A shared counter maintains the number of processes that have arrived at the barrier, and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, the process checks to see if the counter equals *p*, i.e. if it is the last process to have arrived. If not, it busy waits on the flag associated with the barrier; if so, it writes the flag to release the waiting processes. A simple barrier algorithm may therefore look like:

```
struct bar_type {
     int counter;
     struct lock_type lock;
     int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
     LOCK(bar_name.lock);
     if (bar_name.counter == 0)
         bar_name.flag = 0;              /* reset flag if first to reach*/
     mycount = bar_name.counter++;   /* mycount is a private variable*/
     UNLOCK(bar_name.lock);
     if (mycount == p) {              /* last to arrive */
         bar_name.counter = 0;       /* reset counter for next barrier */
         bar_name.flag = 1;          /* release waiting processes */
     }
     else
         while (bar_name.flag == 0) {};/* busy wait for release */
}
```

Centralized Barrier with Sense Reversal

Can you see a problem with the above barrier? There is one. It occurs when the same barrier (barrier bar_name above) is used consecutively. That is, each processor executes the following code:

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

The first process to enter the barrier the second time re-initializes the barrier counter, so that is not a problem. The problem is the flag. To exit the first barrier, processes spin on the flag until it is set to 1. Processes that see flag change to one will exit the barrier, perform the subsequent computation, and enter the barrier again. Suppose one processor P_x gets stuck while in the spin loop; for example, it gets swapped out by the operating system because it has been spinning too long. When it is swapped back in, it will continue to wait for the flag to change to 1. However, in the meantime other processes may have entered the second instance of the barrier, and the first of these will have reset the flag to 0. Now the flag will only get set to 1 again when all p processes have registered at the new instance of the barrier, which will never happen since P_x will never leave the spin loop and get to this barrier instance.

How can we solve this problem? What we need to do is prevent a process from entering a new instance of a barrier before all processes have exited the previous instance of the same barrier. One way is to use another counter to count the processes that leave the barrier, and not let a process enter a new barrier instance until this counter has turned to p for the previous instance. However, manipulating this counter incurs further latency and contention. A better solution is the following. The main reason for the problem in the previous case is that the flag is reset before all processes reach the next instance of the barrier. However, with the current setup we clearly cannot wait for all processes to reach the barrier before resetting the flag, since that is when we actually set the flag for the release. The solution is to have processes wait for the flag to obtain a different value in consecutive instances of the barrier, so for example processes may wait for the flag to turn to 1 in one instance and to turn to 0 in the next instance. A private variable is used per process to keep track of which value to wait for in the current barrier instance. Since by the semantics of a barrier a process cannot get more than one barrier ahead of another, we only need two values (0 and 1) that we toggle between each time, so we call this method sense-reversal. Now the flag need not be reset when the first process reaches the barrier; rather, the process stuck in the old barrier instance still waits for the flag to reach the old release value (sense), while processes that enter the new instance wait for the other (toggled) release value. The value of the flag is only changed when all processes have reached the barrier instance, so it will not change before processes stuck in the old instance see it. Here is the code for a simple barrier with sense-reversal.

```
BARRIER (bar_name, p)
{
    local_sense = !(local_sense);        /* toggle private sense variable */
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;        /* mycount is a private variable*/
    if (bar_name.counter == p) {         /* last to arrive */
        UNLOCK(bar_name.lock);
        bar_name.counter = 0;            /* reset counter for next barrier */
        bar_name.flag = local_sense;     /* release waiting processes */
    }
    else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {};/* busy wait for release */
    }
}
```

The lock is not released immediately after the increment of the counter, but only after the condition is evaluated; the reason for this is left as an exercise (see Exercise 5.7) We now have a correct barrier that can be reused any number of times consecutively. The remaining issue is performance, which we examine next. (Note that the LOCK/UNLOCK protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation.

Performance Issues

The major performance goals for a barrier are similar to those for locks:

Low latency (small critical path length) Ignoring contention, we would like the number of operations in the chain of dependent operations needed for *p* processors to pass the barrier to be small.

Low traffic Since barriers are global operations, it is quite likely that many processors will try to execute a barrier at the same time. We would like the barrier algorithm to reduce the number of bus transactions and hence the possible contention.

Scalability Related to traffic, we would like to have a barrier that scales well to the number of processors we may want to support.

Low storage cost We would of course like to keep the storage cost low.

Fairness This is not much of an issue since all processors get released at about the same time, but we would like to ensure that the same processor does not always become the last one to exit the barrier, or to preserve FIFO ordering.

In a centralized barrier, each processor accesses the lock once, so the critical path length is at least *p*. Consider the bus traffic. To complete its operation, a centralized barrier involving *p* processors performs *2p* bus transactions to get the lock and increment the counter, two bus transactions for the last processor to reset the counter and write the release flag, and another *p-1* bus transactions to read the flag after it has been invalidated. Note that this is better than the traffic for

even a test-and-test&set lock to be acquired by *p* processes, because in that case each of the *p* releases causes an invalidation which results in *O(p)* processes trying to perform the test&set again, thus resulting in $O(p^2)$ bus transactions. However, the contention resulting from these competing bus transactions can be substantial if many processors arrive at the barrier simultaneously, and barriers can be expensive.

Improving Barrier Algorithms for a Bus

Let us see if we can devise a better barrier for a bus. One part of the problem in the centralized barrier is that all processors contend for the same lock and flag variables. To address this, we can construct barriers that cause less processors to contend for the same variable. For example, processors can signal their arrival at the barrier through a software combining tree (see Chapter 3, Section 3.4.2). In a binary combining tree, only two processors notify each other of their arrival at each node of the tree, and only one of the two moves up to participate at the next higher level of the tree. Thus, only two processors access a given variable. In a network with multiple parallel paths, such as those found in larger-scale machines, a combining tree can perform much better than a centralized barrier since different pairs of processors can communicate with each other in different parts of the network in parallel. However, with a centralized interconnect like a bus, even though pairs of processors communicate through different variables they all generate transactions and hence contention on the same bus. Since a binary tree with *p* leaves has approximately *2p* nodes, a combining tree requires a similar number of bus transactions to the centralized barrier. It also has higher latency since it requires *log p* steps to get from the leaves to the root of the tree, and in fact on the order of *p* serialized bus transactions. The advantage of a combining tree for a bus is that it does not use locks but rather simple read and write operations, which may compensate for its larger uncontended latency if the number of processors on the bus is large. However, the simple centralized barrier performs quite well on a bus, as shown in Figure 5-31. Some of the other, more scalable barriers shown in the figure for illustration will be discussed, along with tree barriers, in the context of scalable machines in Chapter 7.
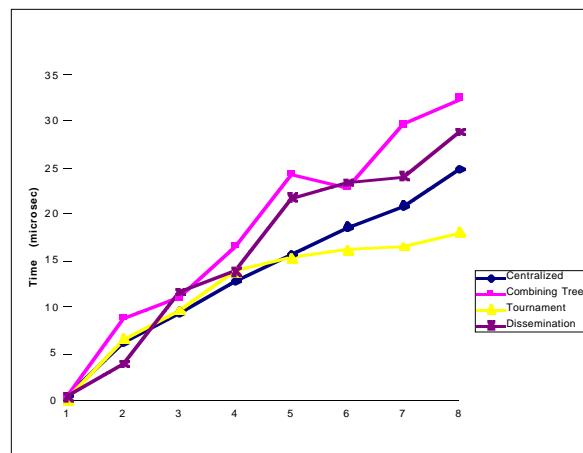
Figure 5-31 Performance of some barriers on the SGI Challenge.

Performance is measured as average time per barrier over a loop of many consecutive barriers. The higher critical path latency of the combining barrier hurts it on a bus, where it has no traffic and contention advantages.

Hardware Primitives

Since the centralized barrier uses locks and ordinary reads/writes, the hardware primitives needed depend on what lock algorithms are used. If a machine does not support atomic primitives well, combining tree barriers can be useful for bus-based machines as well.

A special bus primitive can be used to reduce the number of bus transactions in the centralized barrier. This optimization takes advantage of the fact that all processors issue a read miss for the same value of the flag when they are invalidated at the release. Instead of all processors issuing a separate read miss bus transaction, a processor can monitor the bus and abort its read miss before putting it on the bus if it sees the response to a read miss to the same location (issued by another processor that happened to get the bus first), and simply take the return value from the bus. In the best case, this "piggybacking" can reduce the number of read miss bus transactions from p to one.

Hardware Barriers

If a separate synchronization bus is provided, it can be used to support barriers in hardware too. This takes the traffic and contention off the main system bus, and can lead to higher-performance barriers. Conceptually, a wired-AND is enough. A processor sets its input to the gate high when it reaches the barrier, and waits till the output goes high before it can proceed. (In practice, reusing barriers requires that more than a single wire be used.) Such a separate hardware mechanism for barriers is particularly useful if the frequency of barriers is very high, as it may be in programs that are automatically parallelized by compilers at the inner loop level and need global synchronization after every innermost loop. However, it can be difficult to manage when not all processors on the machine participate in the barrier. For example, it is difficult to dynamically change the number of processors participating in the barrier, or to adapt the configuration when processes are migrated among processors by the operating system. Having multiple participating processes per processor also causes complications. Current bus-based multiprocessors therefore do not tend to provide special hardware support, but build barriers in software out of locks and shared variables.

5.6.6 Synchronization Summary

While some bus-based machines have provided full hardware support for synchronization operations such as locks and barriers, issues related to flexibility and doubts about the extent of the need for them has led to a movement toward providing simple atomic operations in hardware and synthesizing higher level synchronization operations from them in software libraries. The programmer can thus be unaware of these low-level atomic operations. The atomic operations can be implemented either as single instructions, or through speculative read-write instruction pairs like load-locked and store-conditional. The greater flexibility of the latter is making them increasingly popular. We have already seen some of the interplay between synchronization primitives, algorithms, and architectural details. This interplay will be much more pronounced when we discuss synchronization for scalable shared address space machines in the coming chapters. Theoretical research has identified the properties of different atomic exchange operations in terms of the time complexity of using them to implement synchronized access to variables. In particular, it is found that simple operations like test&set and fetch&op are not powerful enough to guarantee that the time taken by a processor to access a synchronized variable is independent of the number

of processors, while more sophisticated atomic operations like compare&swap and an memory-to-memory swap are [Her91].

5.7 Implications for Software

So far, we have looked at architectural issues and how architectural and protocol tradeoffs are affected by workload characteristics. Let us now come full circle and examine how the architectural characteristics of these small-scale machines influence parallel software. That is, instead of keeping the workload fixed and improving the machine or its protocols, we keep the machine fixed and examine how to improve parallel programs. Improving synchronization algorithms to reduce traffic and latency was an example of this, but let us look at the parallel programming process more generally, particularly the data locality and artifactual communication aspects of the orchestration step.

Of the techniques discussed in Chapter 3, those for load balance and inherent communication are the same here as in that general discussion. In addition, one general principle that is applicable across a wide range of computations is to try to assign computation such that as far as possible only one processor writes a given set of data, at least during a single computational phase. In many computations, processors read one large shared data structure and write another. For example, in Raytrace processors read a scene and write an image. There is a choice of whether to partition the computation so the processors write disjoint pieces of the destination structure and read-share the source structure, or so they read disjoint pieces of the source structure and write-share the destination. All other considerations being equal (such as load balance and programming complexity), it is usually advisable to avoid write-sharing in these situations. Write-sharing not only causes invalidations and hence cache misses and traffic, but if different processes write the same words it is very likely that the writes must be protected by synchronization such as locks, which are even more expensive.

The structure of communication is not much of a variable: With a single centralized memory, there is little incentive to use explicit large data transfers, so all communication is implicit through loads and stores which lead to the transfer of cache blocks. DMA can be used to copy large chunks of data from one area of memory to another faster than loads and stores, but this must be traded off against the overhead of invoking DMA and the possibility of using other latency hiding techniques instead. And with a zero-dimensional network topology (a bus) mapping is not an issue, other than to try to ensure that processes migrate from one processor to another as little as possible, and is invariably left to the operating system. The most interesting issues are related to temporal and spatial locality: to reduce the number of cache misses, and hence reduce both latency as well as traffic and contention on the shared bus.

With main memory being centralized, temporal locality is exploited in the processor caches. The specialization of the working set curve introduced in Chapter 3 is shown in Figure 5-32. All capacity-related traffic goes to the same local bus and centralized memory. The other three kinds of misses will occur and generate bus traffic even with an infinite cache. The major goal is to have working sets fit in the cache hierarchy, and the techniques are the same as those discussed in Chapter 3.

For spatial locality, a centralized memory makes data distribution and the granularity of allocation in main memory irrelevant (only interleaving data among memory banks to reduce conten-
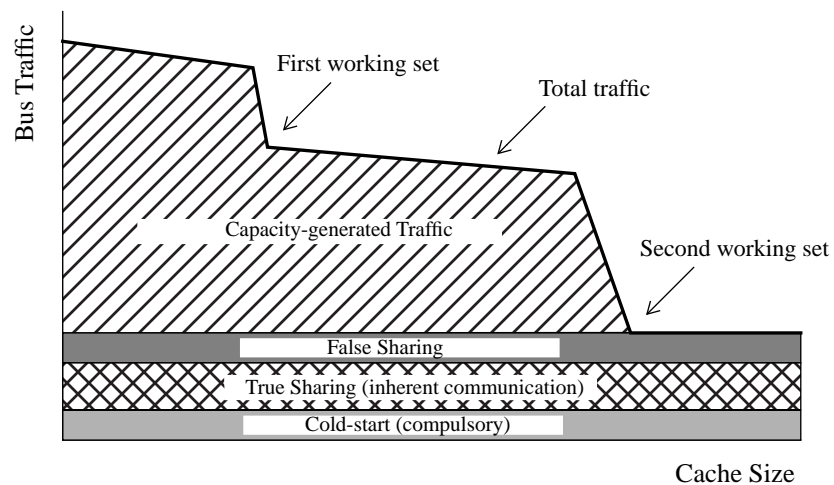
Figure 5-32 Data traffic on the shared bus, and its components as a function of cache size.

The points of inflection indicate the working sets of the program.

tion may be an issue just as in uniprocessors). The ill effects of poor spatial locality are *fragmentation*, i.e. fetching unnecessary data on a cache block, and *false sharing*. The reasons are that the *granularity of communication* and the *granularity of coherence* are both cache blocks, which are larger than a word. The former causes fragmentation in communication or data transfer, and the latter causes false sharing (we assume here that techniques like subblock dirty bits are not used, since they are not found in actual machines). Let us examine some techniques to alleviate these problems and effectively exploit the prefetching effects of long cache blocks, as well as techniques to alleviate cache conflicts by better spatial organization of data. There are many such techniques in a programmer's "bag of tricks", and we only provide a sampling of the most general ones.

Assign tasks to reduce spatial interleaving of access patterns. It is desirable to assign tasks such that each processor tends to access large contiguous chunks of data. For example, if an array computation with *n* elements is to be divided among *p* processors, it is better to divide it so that each processor accesses *n/p* contiguous elements rather than use a finely interleaved assignment of elements. This increases spatial locality and reduces false sharing of cache blocks. Of course, load balancing or other constraints may force us to do otherwise.

Structure data to reduce spatial interleaving of access patterns. We saw an example of this in the equation solver kernel in Chapter 3, when we used higher-dimensional arrays to keep a processor's partition of an array contiguous in the address space. There, the motivation was to allow proper distribution of data among physical memories, which is not an issue here. However, the same technique also helps reduce false sharing, fragmentation of data transfer, and conflict misses as shown in Figures 5-33 and 5-34, all of which cause misses and traffic on the bus. Consider false sharing and fragmentation for the equation solver kernel and the Ocean application. A cache block larger than a single grid element may straddle a column-oriented partition boundary as shown in Figure 5-33(a). If the block is larger than two grid elements, as is likely, it can cause communication due to false sharing. It is easiest to see if we assume for a moment that there is no inherent communication in the algorithm; for example, suppose in each sweep a process simply

(a) Two-dimensional array   (a) Three-dimensional array
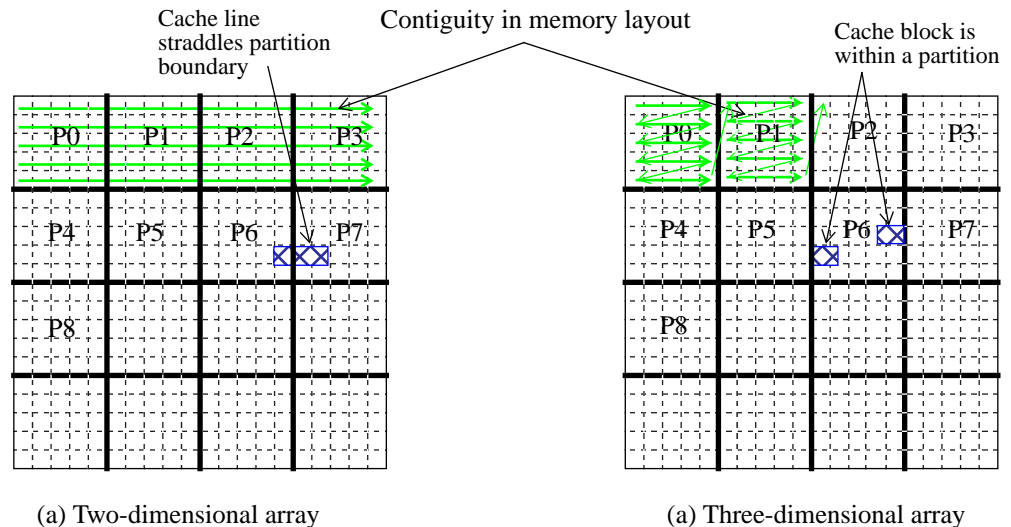
Figure 5-33 Reducing false sharing and fragmentation by using higher-dimensional arrays to keep partitions contiguous in the address space.

In the two-dimensional array case, cache blocks straddling partition boundaries cause both fragmentation (a miss brings in data from the other processor's partition) as well as false sharing.

added a constant value to each of its assigned grid elements, instead of performing a nearest-neighbor computation. A cache block straddling a partition boundary would be false-shared as different processors wrote different words on it. This would also cause fragmentation in communication, since a process reading its own boundary element and missing on it would also fetch other elements in the other processor's partition that are on the same cache block, and that it does not need. The conflict misses problem is explained in Figure 5-34. The issue in all these cases is also non-contiguity of partitions, only at a finer granularity (cache line) than for allocation (page). Thus, a single data structure transformation helps us solve all our spatial locality related problems in the equation solver kernel. Figure 5-35 illustrates the impact of using higher-dimensional arrays to represent grids or blocked matrices in the Ocean and LU applications on the SGI Challenge, where data distribution in main memory is not a issue. The impact of conflicts and false sharing on uniprocessor and multiprocessor performance is clear.

Beware of conflict misses. Figure 5-33 illustrates how allocating power of two sized arrays can cause cache conflict problems—which can become quite pathological in some cases—since the cache size is also a power of two. Even if the logical size of the array that the application needs is a power of two, it is often useful to allocate a larger array that is not a power of two and then access only the amount needed. However, this can interfere with allocating data at page granularity (also a power of two) in machines with physically distributed memory, so we may have to be careful. The cache mapping conflicts in these array or grid examples are within a data structure accessed in a predictable manner, and can thus be alleviated in a structured way. Mapping conflicts are more difficult to avoid when they happen *across* different major data structures (e.g. across different grids used by the Ocean application), where they may have to be alleviated by ad hoc padding and alignment. However, they are particularly insidious in a shared address space when they occur on seemingly harmless shared variables or data structures that a programmer is not inclined to think about, but that can generate artifactual communication. For example, a fre-

Locations in subrows ⟍⟍⟍⟍ and ⟋⟋⟋⟋ map to the same entries (indices) in the cache. The rest of the processor's cache entries are not mapped to by locations in its partition (but would have been mapped to by subrows ▭ in other processors' partitions) and are thus wasted.
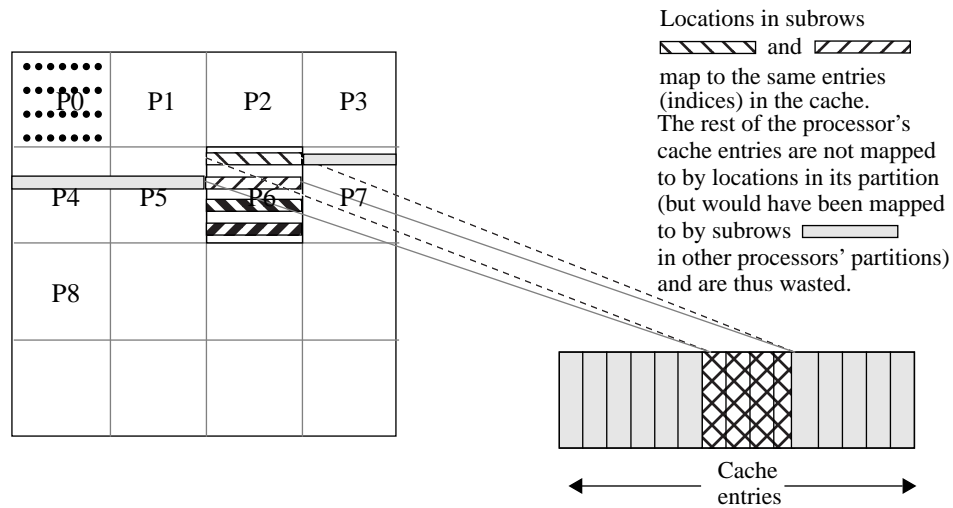
Figure 5-34 Cache mapping conflicts caused by a two-dimensional array representation in a direct-mapped cache.

The figure shows the worst case, in which the separation between successive subrows in a process's partition (i.e. the size of a full row of the 2-d array) is exactly equal to the size of the cache, so consecutive subrows map directly on top of one another in the cache. Every subrow accessed knocks the previous subrow out of the cache. In the next sweep over its partition, the processor will miss on every cache block it references, even if the cache as a whole is large enough to fit a whole partition. Many intermediately poor cases may be encountered depending on grid size, number of processors and cache size. *Since the cache size in bytes is a power of two, sizing the dimensions of allocated arrays to be powers of two is discouraged.*
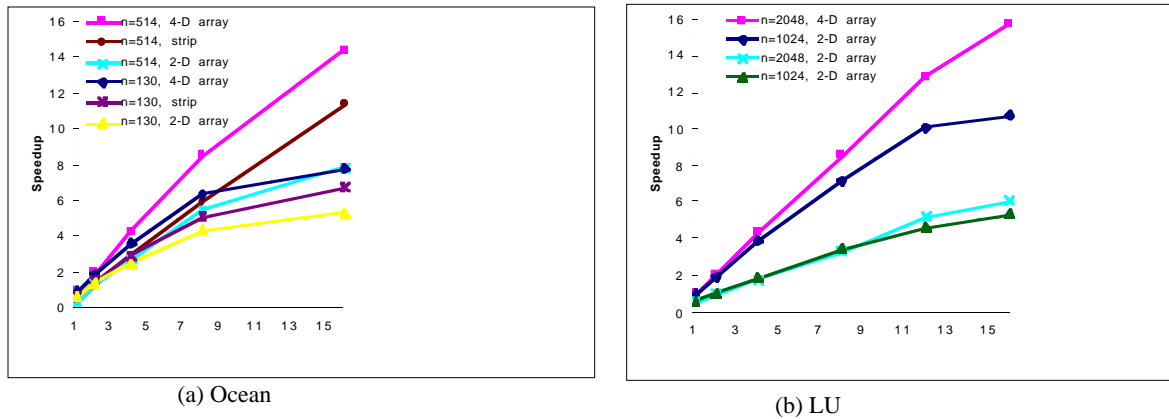
Figure 5-35 Effects of using 4-D versus 2-D arrays to represent two-dimensional grid or matrix data structures on the SGI Challenge.

Results are shown for different problem sizes for the Ocean and LU applications.

quently accessed pointer to an important data structure may conflict in the cache with a scalar variable that is also frequently accessed during the same computation, causing a lot of traffic. Fortunately, such problems tend to be infrequent in modern caches. In general, efforts to exploit locality can be wasted if attention is not paid to reducing conflict misses.

Use per-processor heaps: It is desirable to have separate heap regions for each processor (or process) from which it allocates data dynamically. Otherwise, if a program performs a lot of very small memory allocations, data used by different processors may fall on the same cache block.

Copy data to increase locality: If a processor is going to reuse a set of data that are otherwise allocated non-contiguously in the address space, it is often desirable to make a contiguous copy of the data for that period to improve spatial locality and reduce cache conflicts. Copying requires memory accesses and has a cost, and is not useful if the data are likely to reside in the cache anyway. But otherwise the cost can be overcome by the advantages. For example, in blocked matrix factorization or multiplication, with a 2-D array representation of the matrix a block is not contiguous in the address space (just like a partition in the equation solver kernel); it is therefore common to copy blocks used from another processor's assigned set to a contiguous temporary data structure to reduce conflict misses. This incurs a cost in copying, which must be traded off against the benefit of reducing conflicts. In particle-based applications, when a particle moves from one processor's partition to another, spatial locality can be improved by moving the attributes of that particle so that the memory for all particles being handled by a processor remains contiguous and dense.

Pad arrays: Beginning parallel programmers often build arrays that are indexed using the process identifier. For example, to keep track of load balance, one may have an array each entry of which is an integer that records the number of tasks completed by the corresponding processor. Since many elements of such an array fall into a single cache block, and since these elements will be updated quite often, false sharing becomes a severe problem. One solution is to pad each entry with dummy words to make its size as large as the cache block size (or, to make the code more robust, as big as the largest cache block size on anticipated machines), and then align the array to a cache block. However, padding many large arrays can result in significant waste of memory, and can cause fragmentation in data transfer. A better strategy is to combine all such variables for a process into a record, pad the entire record to a cache block boundary, and create an array of records indexed by process identifier.

Determine how to organize arrays of records: Suppose we have a number of logical records to represent, such as the particles in the Barnes-Hut gravitational simulation. Should we represent them as a single array of n particles, each entry being a record with fields like position, velocity, force, mass, etc. as in Figure 5-36(a)? Or should we represent it as separate arrays of size n, one per field, as in Figure 5-36(b)? Programs written for vector machines, such as traditional Cray computers, tend to use a separate array for each property of an object, in fact even one per field per physical dimension x, y or z. When data are accessed by field, for example the x coordinate of all particles, this increases the performance of vector operations by making accesses to memory unit stride and hence reducing memory bank conflicts. In cache-coherent multiprocessors, however, there are new tradeoffs, and the best way to organize data depends on the access patterns. Consider the update phase of the Barnes-Hut application. A processor reads and writes only the position and velocity fields of all its assigned particles, but its assigned particles are not contiguous in the shared particle array. Suppose there is one array of size n (particles) per field or property. A double-precision three-dimensional position (or velocity) is 24 bytes of data, so several of these may fit on a cache block. Since adjacent particles in the array may be read and written by other processors, poor spatial locality and false sharing can result. In this case it is better to have a single array of particle records, where each record holds all information about that particle.

Now consider the force-calculation phase of the same Barnes-Hut application. Suppose we use an organization by particle rather than by field as above. To compute the forces on a particle, a

Particle 0   Particle 1   · · ·   Particle n

x | y | z | v_x|v_y|v_z|f_x|f_y|f_z | | | | | | | | | | | | | x | y | z | v_x|v_y|v_z|f_x|f_y|f_z

(a) Organization by particle

P 0   P1   · · ·   P n

position: x | y | z | x | y | z | | | x | y | z

velocity: v_x|v_y|v_z|v_x|v_y|v_z| | | |v_x|v_y|v_z

force: f_x|f_y|f_z|f_x|f_y|f_z| | | f_x|f_y|f_z
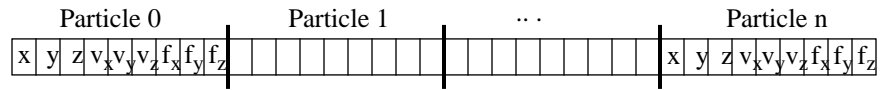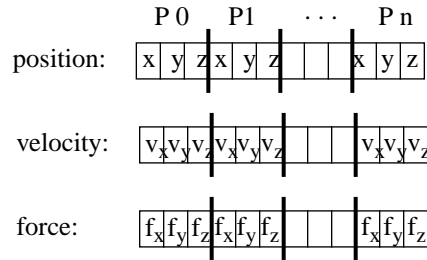
(a) Organization by property or field

Figure 5-36 Alternative data structure organizations for record-based data.

processor reads the position values of many other particles and cells, and it then updates the force components of its own particle. In updating force components, it invalidates the position values of this particle from the caches of other processors that are using and reusing them, due to false sharing, even though the position values themselves are not being modified in this phase of computation. In this case, it would probably be better if we were to split the single array of particle records into two arrays of size n each, one for positions (and perhaps other fields) and one for forces. The force array itself could be padded to reduce false-sharing in it. In general, it is often beneficial to split arrays to separate fields that are used in a read-only manner in a phase from those fields whose values are updated in the same phase. Different situations dictate different organizations for a data structure, and the ultimate decision depends on which pattern or phase dominates performance.

Align arrays: In conjunction with the techniques discussed above, it is often necessary to align arrays to cache block boundaries to achieve the full benefits. For example, given a cache block size of 64 bytes and 8 byte fields, we may have decided to build a single array of particle records with x, y, z, fx, fy, and fx fields, each 48-byte record padded with two dummy fields to fill a cache block. However, this wouldn't help if the array started at an offset of 32 bytes from a page in the virtual address space, as this would mean that the data for each particle will now span two cache blocks. Alignment is easy to achieve by simply allocating a little extra memory and suitably adjusting the starting address of the array.

As seen above, the organization, alignment and padding of data structures are all important to exploiting spatial locality and reducing false sharing and conflict misses. Experienced programmers and even some compilers use these techniques. As we discussed in Chapter 3, these locality and artifactual communication issues can be significant enough to overcome differences in inherent communication, and can cause us to revisit our algorithmic partitioning decisions for an application (recall strip versus subblock decomposition for the simple equation solver as discussed in Chapter 3, Section 3.2.2).

5.8 Concluding Remarks

Symmetric shared-memory multiprocessors are a natural extension of uniprocessor workstations and personal-computers. Sequential applications can run totally unchanged, and yet benefit in performance by getting a larger fraction of a processor's time and from the large amount of shared main-memory and I/O capacity typically available on such machines. Parallel applications are also relative easy to bring up, as all local and shared data are directly accessible from all processors using ordinary loads and stores. Computationally intensive portions of a sequential application can be selectively parallelized. A key advantage for multiprogrammed workloads is the fine granularity at which resources can be shared among application processes. This is true both temporally, in that processors and/or main-memory pages can be frequently reallocated among different application processes, and also physically, in that main-memory may be split among applications at the granularity of an individual page. Because of these appealing features, all major vendors of computer systems, from workstation suppliers like SUN, Silicon Graphics, Hewlett Packard, Digital, and IBM to personal computer suppliers like Intel and Compaq are producing and selling such machines. In fact, for some of the large workstation vendors, these multiprocessors constitute a substantial fraction of their revenue stream, and a still larger fraction of their net profits due to the higher margins [Hep90] on these higher-end machines.

The key technical challenge in the design of such machines is the organization and implementation of the shared memory subsystem, which is used for communication between processors in addition to handling all regular memory accesses. The majority of small-scale machines found today use the system bus as the interconnect for communication, and the challenge then becomes how to keep shared data in the private caches of the processors coherent. There are a large variety of options available to the system architect, as we have discussed, including the set of states associated with cache blocks, choice of cache block size, and whether updates or invalidates are used. The key task of the system architect is to make choices that will perform well based on predominant sharing patterns expected in workloads, and those that will make the task of implementation easier.

As processor, memory-system, integrated-circuit, and packaging technology continue to make rapid progress, there are questions raised about the future of small-scale multiprocessors and importance of various design issues. We can expect small-scale multiprocessors to continue to be important for at least three reasons. The first is that they offer an attractive cost-performance point. Individuals or small groups of people can easily afford them for use as a shared resource or a compute- or file-server. Second, microprocessors today are designed to be multiprocessor-ready, and designers are aware of future microprocessor trends when they begin to design the next generation multiprocessor, so there is no longer a significant time lag between the latest microprocessor and its incorporation in a multiprocessor. As we saw in Chapter 1, the Intel Pentium processor line plugs in "gluelessly" into a shared bus. The third reason is that the essential software technology for parallel machines (compilers, operating systems, programming languages) is maturing rapidly for small-scale shared-memory machines, although it still has a substantial way to go for large-scale machines. Most computer systems vendors, for example, have parallel versions of their operating systems ready for their bus-based multiprocessors. The design issues that we have explored in this chapter are fundamental, and will remain important with progress in technology. This is not to say that the optimal design choices will not change.

We have explored many of the key design aspects of bus-based multiprocessors at the "logical level" involving cache block state transitions and complete bus transactions. At this level the

approach appears to be a rather simple extension of traditional cache controllers. However, much of the difficulty in such designs, and many of the opportunities for optimization and innovation occur at the more detailed "physical level." The next chapter goes down a level deeper into the hardware design and organization of bus-based cache-coherent multiprocessors and some of the natural generalizations.

5.9 References

[AdG96]   Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. IEEE Computer, vol. 29, no. 12, December 1996, pp. 66-76.

[AgG88]   Anant Agarwal and Anoop Gupta. Memory-reference Characteristics of Multiprocessor Applications Under MACH. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, May 1988.

[ArB86]   James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[BaW88]   Jean-Loup Baer and Wen-Hann Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on   Computer Architecture*, pp. 73--80, May 1988.

[BJS88]   F. Baskett, T. Jermoluk, and D. Solomon, The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. *Proc of the 33rd IEEE Computer Society Int. Conf. - COMPCOM 88*, pp. 468-71, Feb. 1988.

[BRG+89]   David Black, Richard Rashid, David Golub, Charles Hill, Robert Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[CAH+93]   Ken Chan, et al. Multiprocessor Features of the HP Corporate Business Servers. In *Proceedings of COMPCON*, pp. 330-337, Spring 1993.

[CoF93]   Alan Cox and Robert Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 98-108, May 1993.

[Dah95]   Fredrik Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 60-69.

[DDS94]   Fredrik Dahlgren and Michel Dubois and Per Stenstrom. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp. 187-197.

[DSB86]   Michel Dubois, Christoph Scheurich and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 434-442.

[DSR+93]   Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 88-97, May 1993.

[DuL92]   C. Dubnicki and T. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 170-180, May 1992.

[EgK88]   Susan Eggers and Randy Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Sym-*

posium on Computer Architecture, pp. 373-382, May 1988.

[EgK89a]   Susan Eggers and Randy Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270, May 1989.

[EgK89b]   Susan Eggers and Randy Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 2-15, May 1989.

[FCS+93]   Jean-Marc Frailong, et al. The Next Generation SPARC Multiprocessing System Architecture. In *Proceedings of COMPCON*, pp. 475-480, Spring 1993.

[GaW93]   Mike Galles and Eric Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In Proceedings of 27th Annual Hawaii International Conference on Systems Sciences, January 1993.

[Goo83]   James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on   Computer Architecture*, pp. 124-131, June 1983.

[Goo87]   James Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987.

[GrCh96]   M. Greenwald and D.R. Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure, Proceedings of the Second Symposium on Operating System Design and Implementation. USENIX, Seattle, October, 1996, pp 123-136.

[GSD95]   H. Grahn, P. Stenstrom, and M. Dubois, Implementation and Evaluation of Update-based Protocols under Relaxed Memory Consistency Models. In Future Generation Computer Systems, 11(3): 247-271, June 1995.

[GrT90]   Gary Granuke and Shreekant Thakkar. Synchronization Algorithms for Shared Memory Multiprocessors. IEEE Computer, vol 23, no. 6, pp. 60-69, June 1990.

[HeP90]   John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 1990.

[Her88]   Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. Seventh ACM SIGACTS-SIGOPS Symposium on Principles of Distributed Computing, August 88.

[Her93]   M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects ACM Transactions on Programming Languages and Systems, 15(5), 745-770, November, 1993

[HeWi87]   Maurice Herlihy and Janet Wing. Axioms for Concurrent Objects, Proc. of the 14th ACM Symp. on Principles of Programming Languages, pp. 13-26, Jan. 1987.

[HiS87]   Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. IEEE Transactions on Computers, vol. C-38, no. 12, December 1989, pp. 1612-1630.

[HEL+86]   Mark Hill et al. Design Decisions in SPUR. *IEEE Computer*, 19(10):8-22, November 1986.

[HeMo93]   M.P. Herlihy and J.E.B. Moss. Transactional Memory: Architectural support for lock-free data structures. 1993 20th Annual Symposium on Computer Architecture San Diego, Calif. pp. 289-301. May 1993.

[JeE91]   Tor E. Jeremiassen and Susan J. Eggers. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 377-381.

[KMR+86]   A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator. Competitive Snoopy Caching. In Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986.

[Kro81]   D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pp. 81-87, May 1981.

[Lam79]   Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, vol. C-28, no. 9, September 1979, pp. 690-691.

[Lau94]   James Laudon. Architectural and Implementation Tradeoffs for Multiple-Context Processors. Ph.D. Thesis, Computer Systems Laboratory, Stanford University, 1994.

[Lei92]   C. Leiserson, et al. The Network Architecture of the Connection Machine CM-5. Symposium of Parallel Algorithms and Architectures, 1992.

[LLJ+92]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. IEEE Transactions on Parallel and Distributed Systems, 4(1):41-61, January 1993.

[MaPu91] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.

[McC84]   McCreight, E. The Dragon Computer System: An Early Overview. Technical Report, Xerox Corporation, September 1984.

[MCS91]   John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared Memory Mutiprocessors. ACM Transactions on Computer Systems, vol. 9, no. 1, pp. 21-65, February 1991.

[MiSc96]   M. Michael and M. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, pp 267-276, May 1996.

[Mip91]   MIPS R4000 User's Manual. MIPS Computer Systems Inc. 1991.

[PaP84]   Mark Papamarcos and Janak Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348-354, June 1984.

[Papa79]   C. H. Papadimitriou. The Serializability of Conccurent Database Updates, Journal of the ACM, 26(4):631-653, Oct. 1979.

[Ros89]   Bryan Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles*, December 1989.

[ScD87]   Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 234-243.

[SFG+93] Pradeep Sindhu, et al. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. In *Proceedings of COMPCON*, pp. 338-344, Spring 1993.

[SHG93]   Jaswinder Pal Singh, John L. Hennessy and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, vol. 26, no. 7, July 1993.

[Smi82]   Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, September 1982.

[SwS86]   Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 414-423, May 1986.

[TaW97]   Andrew S. Tanenbaum and Albert S. Woodhull, Operating System Design and Implementation (Second Edition), Prentice Hall, 1997.

[Tel90]   Patricia Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26-36, June 1990.

[TBJ+88] M. Thompson, J. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Proceedings of USENIX Technical Conference,* February

1988.

[TLS88]   Charles Thacker, Lawrence Stewart, and Edwin Satterthwaite, Jr. Firefly: A Multiprocessor Workstation, IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 909-20.

[TLH94]   Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651-663, June 1994.

[Val95]   J. Valois, Lock-Free Linked Lists Using Compare-and-Swap, Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ont. Canada, pp 214-222, August 20-23, 1995

[WBL89]   Wen-Hann Wang, Jean-Loup Baer and Henry M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In Proceedings of the 16th Annual International Symposium on Computer Architecture, pp. 140-148, June 1989.

[WeS94]   Shlomo Weiss and James Smith. Power and PowerPC. Morgan Kaufmann Publishers Inc. 1994.

[WEG+86]   David Wood, et al. An In-cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 358-365, May 1986.

5.10 Exercises

5.1 Short Answer.

    a. Is the cache-coherence problem an issue with processor-level registers as well? Given that registers are not kept consistent in hardware, how do current systems guarantee the desired semantics of a program under this hardware model?

    b. Consider the following graph indicating the miss rate of an application as a function of cache block size. As might be expected, the curve has a U-shaped appearance. Consider the three points A, B, and C on the curve. Indicate under what circumstances, if any, each may be a sensible operating point for the machine (i.e. the machine might give better performance at that point rather than at the other two points).
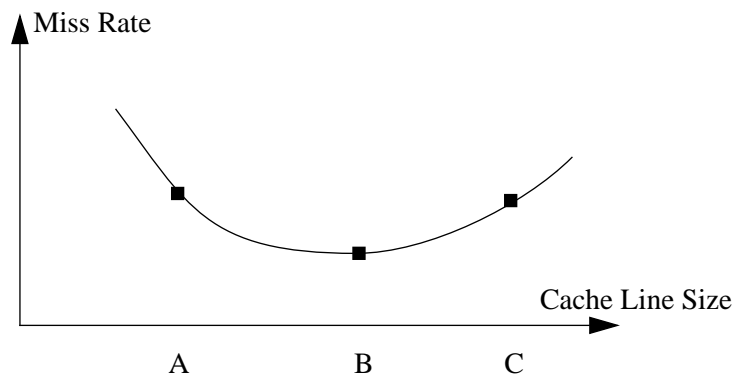
5.2 Bus Traffic

Assume the following average data memory traffic for a bus-based shared memory multiprocessor:

    private reads 70%

    private writes 20%

    shared reads 8%

    shared writes 2%

Also assume that 50% of the instructions (32-bits each) are either loads or stores. With a split instruction/data cache of 32Kbyte total size, we get hit rates of 97% for private data, 95% for shared data and 98.5% for instructions. The cache line size is 16 bytes.

We want to place as many processors as possible on a bus that has 64 data lines and 32 address lines.

The processor clock is twice as fast as that of the bus, and before considering memory penalties the processor CPI is 2.0. How many processors can the bus support without saturating if we use

    (i) write-through caches with write-allocate strategy?

    (ii) write-back caches?

Ignore cache consistency traffic and bus contention. The probability of having to replace a dirty block in the write-back caches is 0.3. For reads, memory responds with data 2 cycles after being presented the address. For writes, both address and data are presented to memory at the same time. Assume that the bus does not have split transactions, and that processor miss penalties are equal to the number of bus cycles required for each miss.

5.3 Update versus invalidate

a. For the memory reference streams given below, compare the cost of executing them on a bus-based machine that (i) supports the Illinois MESI protocol, and (ii) the Dragon protocol. Explain the observed performance differences in terms of the characteristics of the streams and the coherence protocols.

stream-1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3

stream-2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1

stream-3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

In the above streams, all of the references are to the same location: r/w indicates read or write, and the digit refers to the processor issuing the reference. Assume that all caches are initially empty, and use the following cost model: (i) read/write cache-hit: 1 cycle; (ii) misses requiring simple transaction on bus (BusUpgr, BusUpd): 3 cycles, and (iii) misses requiring whole cache block transfer = 6 cycles. Assume all caches are write allocated.

b. As miss latencies increase, does an update protocol become more or less preferable as compared to an invalidate protocol? Explain.

c. One of the key features of the update protocol is that instead of whole cache blocks, individual words are transferred over the bus. Given the design of the SGI Powerpath-2 bus in Section 6.6, estimate the peak available bandwidth if *all* bus transactions were updates. Assume that all updates contain 8 bytes of data, and that they also require a 5-cycle transaction. Then, given the miss-decomposition data for various applications in Figure 5-21, estimate the peak available bus bandwidth on the Powerpath-2 bus for these applications.

d. In a multi-level cache hierarchy, would you propagate updates all the way to the first-level cache or only to the second-level cache? Explain the tradeoffs.

e. Why is update-based coherence not a good idea for multiprogramming workloads typically found on multiprocessor compute servers today?

f. To provide an update protocol as an alternative, some machines have given control of the type of protocol to the software at the granularity of page, that is, a given page can be kept coherent either using an update scheme or an invalidate scheme. An alternative to page-based control is to provide special opcodes for writes that will cause updates rather than invalidates. Comment on the advantages and disadvantages.

5.4 Sequential Consistency.

a. Consider incoming transactions (requests and responses) from the bus down into the cache hierarchy, in a system with two-level caches, and outgoing transactions from the cache hierarchy to the bus. To ensure SC when invalidations are acknowledged early (as soon as they appear on the bus), what ordering constraints must be maintained in each direction among the ones described below, and which ones can be reordered? Now con-

sider the incoming transactions at the first level cache, and answer the question for these. Assume that the processor has only one outstanding request at a time.

Orderings in the upward direction (from bus towards the caches and the processor):

update	invalidate	invalidate	invalidate	invalidate	invalidate
↑	↑	↑	↑	↑	↑
update	update	data reply for load instr.	an explicit completion ack for invalidate or update in distributed memory environments	data request for dirty data in local cache	invalidate
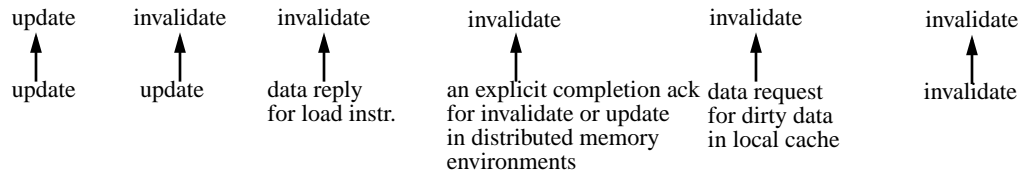
Orderings in the downward direction (from the processor and caches towards the bus):

- processor's single outstanding request can be reordered w.r.t. other replies being generated by the caches;
- replies to requests (where this caches had data in dirty state) can be reordered, since these distinct requests must have been from different processors (as each processor can have only one outstanding load/store request).
- a writeback can be reorderedw.r.t. above data replies for dirty cache blocks.

b. Given the following code segments, say what results are possible (or not possible) under SC. Assume that all variables are initialized to 0 before this code is reached. [

(i)

P1	P2	P3
A = 1	u = A	v = B
	B = 1	w = A

(ii)

P1	P2	P3	P4
A = 1	u = A	B = 1	w = B
	v = B		x = A

(iii) In the following sequence, first consider the operations within a dashed box to be part of the same instruction, say a fetch&increment. Then, suppose they are separate instructions. Answer the questions for both cases.

P1	P2
u = A	v = A
A = u+1	A = v+1

c. Is the reordering problem due to write buffers, discussed in Section 5.3.2, also a problem for concurrent programs on a uniprocessor? If so, how would you prevent it; if not, why not?

d. Can a read complete before a previous write issued by the same processor to the same location has completed (for example, if the write has been placed in the writer's write buffer but has not yet become visible to other processors), and still provide a coherent memory system? If so, what value should the read return? If not, why not? Can this be done and still guarantee sequential consistency?

e. If we cared only about coherence and not about sequential consistency, could we declare a write to complete as soon as the processor is able to proceed past it?

    f.   Are the sufficient conditions for SC necessary? Make them less constraining (i) as much as possible and (ii) in a reasonable intermediate way, and comment on the effects on implementation complexity.

5.5 Consider the following conditions proposed as sufficient conditions for SC:

- Every process issues memory requests in the order specified by the program.
- After a read or write operation is issued, the issuing process waits for the operation to complete before issuing its next operation.
- Before a processor P_j can return a value written by another processor P_i, all operations that were performed with respect to P_i before it issued the store must also be performed with respect to P_j.

Are these conditions indeed sufficient to guarantee SC executions? If not, construct a counter-example, and say why the conditions that were listed in the chapter are indeed sufficient in that case. [Hint: to construct the example, think about in what way these conditions are different from the ones in the chapter. The example is not obvious.]

5.6 Synchronization Performance

    a.   Consider a 4-processor bus-based multiprocessor using the Illinois MESI protocol. Each processor executes a test&set lock to gain access to a null critical section. Assume the test&set instruction always goes on the bus, and it takes the same time as a normal read transaction. Assume that the initial condition is such that processor-1 has the lock, and that processors 2, 3, and 4 are spinning on their caches waiting for the lock to be released. The final condition is that every processor gets the lock once and then exits the program. Considering only the bus transactions related to lock/unlock code:

      (i) What is the least number of transactions executed to get from the initial to the final state?

      (ii) What is the worst case number of transactions?

      (iii) Repeat parts a and b assuming the Dragon protocol.

    b.   What are the main advantages and disadvantages of exponential backoff in locks?

    c.   Suppose all 16 processors in a bus-based machine try to acquire a test-and-test&set lock simultaneously. Assume all processors are spinning on the lock in their caches and are invalidated by a release at time 0. How many bus transactions will it take until all processors have acquired the lock if all the critical sections are empty (i.e. each processor simply does a LOCK and UNLOCK with nothing in between). Assuming that the bus is fair (services pending requests before new ones) and that every bus transaction takes 50 cycles, how long would it take before the *first* processor acquires and releases the lock. How long before the last processor to acquire the lock is able to acquire and release it? What is the best you could do with an unfair bus, letting whatever processor you like win an arbitration regardless of the order of the requests? Can you improve the performance by choosing a different (but fixed) bus arbitration scheme than a fair one?

    d.   If the varaibles used for implementing locks are not cached, will a test&test&set lock still generate less traffic than a test&set lock? Explain your answer.

    e.   For the same machine configuration in the previous exercise with a fair bus, answer the same questions about the number of bus transactions and the time needed for the first and last processors to acquire and release the lock when using a ticket lock. Then do the same for the array-based lock.

    f. For the performance curves for the test&set lock with exponential backoff shown in Figure 5-29 on page 321, why do you think the curve for the non-zero critical section was a little worse than the curve for the null critical section?

    g. Why do we use d to be smaller than c in our lock experiments. What problems might occur in measurement if we used d larger than c. (Hint: draw time-lines for two processor executions.) How would you expect the results to change if we used much larger values for c and d?

5.7 Synchronization Algorithms

    a. Write pseudo code (high-level plus assembly) to implement the ticket lock and array-based lock using (i) fetch&increment, (ii) LL/SC.

    b. Suppose you did not have a fetch&increment primitive but only a fetch&store (a simple atomic exchange). Could you implement the array-based lock with this primitive? Describe the resulting lock algorithm?

    c. Implement a compare&swap operation using LL-SC.

    d. Consider the barrier algorithm with sense reversal that was described in Section 5.6.5. Would there be a problem be if the UNLOCK statement were placed just after the increment of the counter, rather than after each branch of the if condition? What would it be?

    e. Suppose that we have a machinethat supports full/empty bits on every word in hardware. This particular machine allows for the following C-code functions:

ST_Special(locn, val) writes val to data location locn and sets the full bit. If the full bit was already set, a trap is signalled.

int LD_Special(locn) waits until the data location's full bit is set, loads the data, clears the full bit, and returns the data as the result.

Write a C function swap(i,j) that uses the above primitives to atomically swap the contents of two locations A[i] and A[j]. You should allow high concurrency (if multiple PEs want to swap distinct paris of locations, they should be able to do so concurrently). You must avoid deadlock.

    f. The fetch-and-add atomic operation can be used to implement barriers, semaphores and other synchronization mechanisms. The semantics of `fetch-and-add` is such that it returns the value before the addition. Use the fetch-and-add primitive to implement a barrier operation suitable for a shared memory multiprocessor. To use the barrier, a processor must execute: BARRIER(BAR, N), where `N` is the number of processes that need to arrive at the barrier before any of them can proceed. Assume that `N` has the same value in each use of barrier `BAR`. The barrier should be capable of supporting the following code:

```
while (condition) {
   Compute stuff
   BARRIER(BAR, N);
}
```

        1. A proposed solution for implementing the barrier is the following:

```
BARRIER(Var B: BarVariable, N: integer)
{
   if (fetch-and-add(B, 1) = N-1) then
      B := 0;
   else
      while (B <> 0) do {};
}
```

What is the problem with the above code? Write the code for BARRIER in away that avoids the problem.

g. Consider the following implementation of the BARRIER synchronization primitive, used repeatedly within an application, at the end of each phase of computation. Assume that bar.releasing and bar.count are initially zero and bar.lock is initially unlocked. The barrier is described here as a function instead of a macro, for simplicity. Note that this function takes no arguments:

```
struct bar_struct {
   LOCKDEC(lock);
   int count, releasing;
} bar;
...

BARRIER()
{
   LOCK(bar.lock);
   bar.count++;

   if (bar.count == numProcs) {
      bar.releasing = 1;
      bar.count--;
   } else {
      UNLOCK(bar.lock);
      while (! bar.releasing)
         ;
      LOCK(bar.lock);
      bar.count--;
      if (bar.count == 0) {
         bar.releasing = 0;
      }
   }
   UNLOCK(bar.lock);
}
```

(i) This code fails to provide a correct barrier. Describe the problem with this implementation.

(ii) Change the code *as little as possible* so it provides a correct barrier implementation. Either clearly indicate your changes on the code above, or thoroughly describe the changes below.

5.8 Protocol Enhancements. Consider migratory data, which are shared data objects that bounce around among processors, with each processor reading and then writing them. Under the standard MESI protocol, the read miss and the write both generate bus transactions.

a. Given the data in Table 5-1 estimate max bandwidth that can be saved when using upgrades (BusUpgr) instead of BusRdX.

b. It is possible to enhance the state stored with cache blocks and the state-transition diagram, so that such read shortly-followed-by write accesses can be recognized, so that migratory blocks can be directly brought in exclusive state into the cache on the first read miss. Suggest the extra states and the state-transition diagram extensions to achieve above. Using the data in Table 5-1, Table 5-2, and Table 5-3 compute the bandwidth sav-

ings that can be achieved. Are there any other benefits than bandwidth savings? Discuss program situations where the migratory protocol may hurt performance.

c. The Firefly update protocol eliminates the SM state present in the Dragon protocol by suitably updating main memory on updates. Can we further reduce the states in the Dragon and/or Firefly protocols by merging the E and M states. What are the tradeoffs?

d. Instead of using writeback caches in all cases, it has been observed that processors sometimes write only one word in a cache line. To optimize for this case, a protocol has been proposed with the following characteristics:

* On the initial write of a line, the processor writes through to the bus, and accepts the line into the Reserved state.

* On a write for a line in the Reserved state, the line transitions to the dirty state, which uses writeback instead of writethrough.

(i) Draw the state transitions for this protocol, using the INVALID, VALID, RESERVED and MODIFIED states. Be sure that you show an arc for each of BusRead, BusWrite, ProcRead, and ProcWrite for each state. Indicate the action that the processor takes after a slash (e.g. BusWrite/WriteLine). Since both word and line sized writes are used, indicate FlushWord or FlushLine.

(ii) How does this protocol differ from the 4-state Illinois protocol?

(iii) Describe concisely why you think this protocol is not used on a system like the SGI Challenge.

e. On a snoopy bus based cache coherent multiprocessor, consider the case when a processor writes a line shared by many processors (thus invalidating their caches). If the line is subsequently reread by the other processors, each will miss on the line. Researchers have proposed a read-broadcast scheme, in which if one processor reads the line, all other processors with invalid copies of the line read it into their second level cache as well. Do you think this is a good protocol extension? Give at least two reasons to support your choice and at least one that argues the opposite.

5.9 Miss classification Classify the misses in the following reference stream from three processors into the categories shown in Figure 5-20 (follow the format in Table 5-4). Assume that each processor's cache consists of a *single* 4-word cache block.

Seq. No.	P1	P2	P3
1	st w0		st w7
2	ld w6	ld w2	
3		ld w7	
4	ld w2	ld w0	
5		st w2	
6	ld w2		
7	st w2	ld w5	ld w5
8	st w5		
9		ld w3	ld w7
10		ld w6	ld w2
11		ld w2	st w7
12	ld w7		

Seq. No.	P1	P2	P3
13	ld w2		
14		ld w5	
15			ld w2

5.10 Software implications and programming.

a. You are given a bus-based shared-memory machine. Assume that the processors have a cache block size of 32 bytes. Now consider the following simple loop:

```
for i = 0 to 16
for j = 0 to 255 {
A[j] = do_something(A[j]);
}
```

(i) Under what conditions would it be better to use a dynamically-scheduled loop?

(ii) Under what conditions would it be better to use a statically-scheduled loop?

(iii) For a dynamically-scheduled inner loop, how many iterations should a PE pick each time if A is an array of 4-byte integers?

b. You are writing an image processing code, where the image is represented as a 2-d array of pixels. The basic iteration in this computation looks like:

```
for i = 1 to 1024
for j = 1 to 1024
 newA[i,j] = (A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])/4;
```

Assume A is a matrix of 4-byte single precision floats stored in row-major order (i.e A[i,j] and A[i,j+1] are at consecutive addresses in memory). A starts at memory location 0. You are writing this code for a 32 processor machine. Each PE has a 32 Kbyte direct-mapped cache and the cache blocks are 64 bytes.

(i) You first try assigning 32 rows of the matrix to each processor in an interleaved way. What is the actual ratio of computation to bus traffic that you expect? State any assumptions.

(ii) Next you assign 32 contiguous rows of the matrix to each processor. Answer the same questions as above.

(iii) Finally, you use a contiguous assignment of columns instead of rows. Answer the same questions now.

(iv) Suppose the matrix A started at memory location 32 rather than address 0. If we use the same decomposition as in (iii), do you expect this to change the actual ratio of computation to traffic generated in the machine? If yes, will it increase or decrease and why? If not, why not?

c. Consider the following simplified N-body code using an O(N^2) algorithm. Estimate the number of misses per time-step in the steady-state. Restructure the code using techniques suggested in the chapter to increase spatial locality and reduce false-sharing. Try and make your restructuring robust with respect to number of processors and cache block size.

As a baseline, assume 16 processors, 1 Mbyte direct-mapped caches with a 64-byte block size. Estimate the misses for the restructured code. State all assumptions that you make.

```
typedef struct moltype {
     double x_pos, y_pos, z_pos; /* position components */
     double x_vel, y_vel, z_vel; /* velocity components */
     double x_f, y_f, z_f; /* force components */
     } molecule;

#define numMols 4096
#define numPEs 16
molecule mol[numMols]

main()
{
... declarations ...
     for (time=0; time < endTime; time++)
          for (i=myPID; i < numMols; i+=numPEs)
          {
               for (j=0; j < numMols; j++)
               {
                    x_f[i] += x_fn(reads position of mols i & j);
                    y_f[i] += y_fn(reads position of mols i & j);
                    z_f[i] += z_fn(reads position of mols i & j);
               }
               barrier(numPEs);
               for (i=myPID; i < numMols; i += numPEs)
               {
                    writes velocity and position components
                    of mol[i] based on force on mol[i];
               }
               barrier(numPEs);
          }
}
```

CHAPTER 6   Snoop-based Multiprocessor Design

6.1 Introduction

The large differences we see in the performance, cost, and scale of symmetric multiprocessors on the market rest not so much on the choice of the cache coherence protocol, but rather on the design and implementation of the oganizational structure that supports the logical operation of the protocol. The performance impact of protocol trade-offs are well understood and most machines use a variant of the protocols described in the previous chapter. However, the latency and bandwidth that is achieved on a protocol depend on the bus design, the cache design, and the integration with memory, as does the engineering cost of the system. This chapter studies the more detailed physical design issues in snoop-based cache coherent symmetric multiprocessors.

The abstract state transition diagrams for coherence protocols are conceptually simple, however, subtle correctness and performance issues arise at the hardware level. The correctness issues arise mainly because actions that are considered atomic at the abstract level are not necessarily atomic

at the organizational level. The performance issues arise mainly because we want to want to pipeline memory operations and allow many operations to be outstanding, using different components of the memory hierarchy, rather than waiting for each operation to complete before starting the next one. These performance enhancements present further correctness challenges. Modern communication assist design for aggressive cache-coherent multiprocessors presents a set of challenges that are similar in complexity and in form to those of modern processor design, which allows a large number of outstanding instructions and out of order execution.

We need to peel off another layer of the design of snoop-based multiprocessors to understand the practical requirements embodied by state transition diagrams. We must contend with at least three issues. First, the implementation must be correct; second, it should offer high performance, and third, it should require minimal extra hardware. The issues are related. For example, providing high performance often requires that multiple low-level events be outstanding (in progress) at a time, so that their latencies can be overlapped. Unfortunately, it is exactly in these situations—due to the numerous complex interactions between these events—that correctness is likely to be compromised. The product shipping dates for several commercial systems, even for microprocessors that have on-chip coherence controllers, have been delayed significantly because of subtle bugs in the coherence hardware.

Our study begins by enumerating the basic correctness requirements on a cache coherent memory system. A base design, using single level caches and a one transaction at a time atomic bus, is developed in *** and the critical events in processing individual transactions are outlined. We assumes an invalidation protocol for concreteness, but the main issues apply directly to update protocols.*** expands this design to address multilevel cache hierarchies, showing how protocol events propagate up and down the hierarchy. *** expands the base design to utilize a split transaction bus, which allows multiple transactions to be performed in a pipelined fashion, and then brings together multilevel caches and split-transactions. From this design point, it is a small step to support multiple outstanding misses from each processor, since all transactions are heavily pipelined and many take place concurrently. The fundamental underlying challenge throughout is maintaining the illusion of sequential order, as required by the sequential consistency model. Having understood the key design issues in general terms, we are ready to study concrete designs in some detail. *** presents to case studies, the SGI Challenge and the Sun Enterprise, and ties the machine performance back to software trade-offs through a workload driven study of our sample applications. Finally, Section *** examines a number of advanced topics which extend the design techniques in functionality and scale.

6.2 Correctness Requirements

A cache-coherent memory system must, of course, satisfy the requirements of coherence and preserve the semantics dictated by the memory consistency model. In particular, for coherence it should ensure that stale copies are found and invalidated/updated on writes and it should provide write serialization. If sequential consistency is to be preserved by satisfying the sufficient conditions, the design should provide write atomicity and the ability to detect the completion of writes. In addition, it should have the desirable properties of any protocol implementation, including cache coherence, which means it should be free of deadlock and livelock, and should either eliminate starvation or make it very unlikely. Finally, it should cope with error conditions beyond its control (e.g., parity errors), and try to recover from them where possible.

Deadlock occurs when operations are still outstanding but all system activity has ceased. In general, deadlock can potentially arise where multiple concurrent entities incrementally obtain shared resources and hold them in a non-preemptible fashion. The potential for deadlock arises when there is a cycle of resource dependences. A simple analogy is in traffic at an intersection, as shown in Figure 6-1. In the traffic example, the entities are cars and the resources are lanes. Each car needs to acquire two lane resources to proceed through the intersection, but each is holding one.
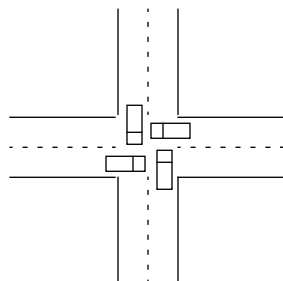
Figure 6-1 Deadlock at a traffic intersection.

Four cars arrive at an intersection and all proceed one lane each into the intersection. They block one another, since each is occupying a resource that another needs to make progress. Even if each yields to the car on the right, the intersection is deadlocked. To break the deadlock, some cars must retreat to allow others to make progress, so that then they themselves can make progress too.

In computer systems, the entities are typically controllers and the resources are buffers. For example, suppose two controllers A and B communicate with each other via buffers, as shown in Figure 6-2(a). A's input buffer is full, and it refuses all incoming requests until B accepts a request from it (thus freeing up buffer space in A to accept requests from other controllers), but B's input buffer is full too, and it refuses all incoming requests until A accepts a request from it. Neither controller can accept a request, so deadlock sets in. To illustrate the problem with more than two controllers, a three-controller example is shown in Figure 6-2(b). It is essential to either avoid such dependence cycles or break them when they occur.
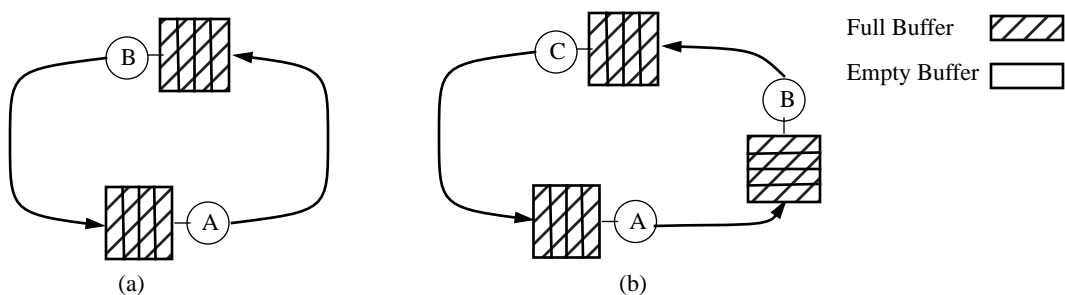
Figure 6-2 Deadlock can easily occur in a system if independent controllers with finite buffering need to communicate with each other.

If cycles are possible in the communication graph, then each controller can be stalled waiting for the one in front to free up resources. We illustrate cases for two and three controllers.

The system is in *livelock* when no processor is making forward progress in its computation even though transactions are being executed in the system. Continuing the traffic analogy, each of the

vehicles might elect to back up, clearing the intersection, and then try again to move forward. However, if they all repeatedly move back and forward at the same time, there will be a lot of activity but they will end up in the same situation repeatedly with no real progress. In computer systems, livelock typically arises when independent controllers compete for a common resource, with one snatching it away from the other before the other has finished with its use for the current operation.

Starvation does not stop overall progress, but is an extreme form of unfairness in which one or more processors make no progress while others continue to do so. In the traffic example, the livelock problem can be solved by a simple priority scheme. If a northbound car is given higher priority than an eastbound car, the latter must pull back and let the former through before trying to move forward again; similarly, a southbound car may have higher priority than a westbound car. Unfortunately, this does not solve starvation: In heavy traffic, an eastbound car may never pass the intersection since there may always be a new northbound car ready to go through. North-bound cars make progress, while eastbound cars are starved. The remedy here is to place an arbiter (e.g. a police officer or traffic light) to orchestrate the resource usage in a fair manner. The analogy extends easily to computer systems.

In general, the possibility of starvation is considered a less catastrophic problem than livelock or deadlock. Starvation does not cause the entire system to stop making progress, and is usually not a permanent state. That is, just because a processor has been starved for some time in the past it does not mean that it will be starved for all future time (at some point northbound traffic will ease up, and eastbound cars will get through). In fact, starvation is much less likely in computer systems than in this traffic example, since it is usually timing dependent and the necessary pathological timing conditions usually do not persist. Starvation often turns out to be quite easy to eliminate in bus-based systems, by having the bus arbitration be fair and using FIFO queues to other hardware resources, rather than rejecting requests and having them be retried. However, in scalable systems that we will see in later chapters, eliminating starvation entirely can add substantial complexity to the protocols and can slow down common-case transactions. Many systems therefore do not completely eliminate starvation, though almost all try to reduce the potential for it to occur.

In the discussions of how cache coherence protocols ensure write serialization and can satisfy the sufficient conditions for sequential consistency, the assumption was made that memory operations are atomic. Here this assumption is made somewhat more realistic: there is a single level of cache per processor and transactions on the bus are atomic. The cache can hold the processor while it performs the series of steps involved in a memory operation, so these operations appear atomic to the processor. The basic issues and tradeoffs that arise in implementing snooping and state transitions are discussed, along with new issues that arise in providing write serialization, detecting write completion, and preserving write atomicity. Then, more aggressive systems are considered, starting with multi-level cache hierarchies, going on to split-transaction (pipelined) buses, and then examining the combination of the two. In split-transaction buses, a bus transaction is split into request and response phases that arbitrate for the bus separately, so multiple transactions can be outstanding at a time on the bus. In all cases, writeback caches are assumed, at least for the caches closest to the bus so they can reduce bus traffic.

6.3 Base Design: Single-level Caches with an Atomic Bus

Several design decisions must be made even for the simple case of single-level caches and an atomic bus. First, how should we design the cache tags and controller, given that both the processor and the snooping agent from the bus side need access to the tags? Second, the results of a snoop from the cache controllers need to be presented as part of the bus transaction; how and when should this be done? Third, even though the memory bus is atomic, the overall set of actions needed to satisfy a processor's request uses other resources as well (such as cache controllers) and is not atomic, introducing possible race conditions. How should we design protocol state machines for the cache controllers given this lack of atomicity? Do these factors introduce new issues with regard to write serialization, write completion detection or write atomicity? And what deadlock, livelock and starvation issues arise? Finally, writebacks from the caches can introduce interesting race conditions as well. The next few subsections address these issues one by one.

6.3.1 Cache controller and tags

Consider first a conventional uniprocessor cache. It consists of a storage array containing data blocks, tags, and state bits, as well as a comparator, a controller, and a bus interface. When the processor performs an operation against the cache, a portion of the address is used to access a cache set that potentially contains the block. The tag is compared against the remaining address bits to determine if the addressed block is indeed present. Then the appropriate operation is performed on the data and the state bits are updated. For example, a write hit to a clean cache block causes a word to be updated and the state to be set to modified. The cache controller sequences the reads and writes of the cache storage array. If the operation requires that a block be transferred from the cache to memory or vice versa, the cache controller initiates a bus operation. The bus operation involves a sequence of steps from the bus interface; these are typically (1) assert request for bus, (2) wait for bus grant, (3) drive address and command, (4) wait for command to be accepted by the relevant device, and (5) transfer data. The sequence of actions taken by the cache controller is itself implemented as a finite state machine, as is the sequencing of steps in a bus transaction. It is important not to confuse these state machines with the transition diagram of the protocol followed by each cache block.

To support a snoopy coherence protocol, the basic uniprocessor cache controller design must be enhanced. First, since the cache controller must monitor bus operations as well as respond to processor operations, it is simplest to view the cache as having two controllers, a bus-side controller and a processor-side controller, each monitoring one kind of external event. In either case, when an operation occurs the controller must access the cache tags. On every bus transaction the bus-side controller must capture the address from the bus and use it to perform a tag check. If the check fails (a snoop miss), no action need be taken; the bus operation is irrelevant to this cache. If the snoop "hits," the controller may have to intervene in the bus transaction according to the cache coherence protocol. This may involve a read-modify-write operation on the state bits and/or trying to obtain the bus to place a block on it.

With only a single array of tags, it is difficult to allow two controllers to access the array at the same time. During a bus transaction, the processor will be locked out from accessing the cache, which will degrade processor performance. If the processor is given priority, effective bus bandwidth will decrease because the snoop controller will have to delay the bus transaction until it

gains access to the tags. To alleviate this problem, many coherent cache designs utilize a *dual-ported* tag and state store or duplicate the tag and state for every block. The data portion of the cache is not duplicated. If tags are duplicated, the contents of the two sets of tags are exactly the same, except one set is used by the processor-side controller for its lookups and the other is used by the bus-side controller for its snoops (see Figure 6-3). The two controllers can read the tags and perform checks simultaneously. Of course, when the tag for a block is updated (e.g. when the state changes on a write or a new block is brought into the cache) both copies must ultimately be modified, so one of the controllers may have to be locked out for a time. For example, if the bus-side tags are updated by a bus transaction, at some point the processor-side tags will have to be updated as well. Machine designs can play several tricks to reduce the time for which a controller is locked out, for example in the above case by having the processor-side tags be updated only when the cache data are later modified, rather than immediately when the bus-side tags are updated. The frequency of tag updates is also much smaller than read accesses, so the coherence snoops are expected to have little impact on processor cache access.

Tags and state used by
the processor
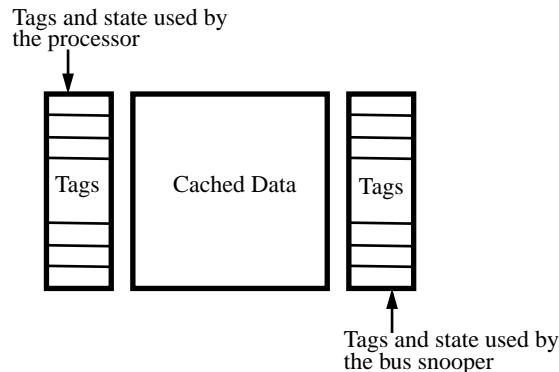
Figure 6-3 Organization of single-level snoopy caches.

For single-level caches there is a duplicate set of tags and state provided to reduce contention. One set is used exclusively by the processor, while another is used by the bus-snooper. Any changes to cache content or state, however, involves updating both sets of tags simultaneously.

Another major enhancement from a uniprocessor cache controller is that the controller now acts not only as an initiator of bus transactions, but also as a responder to them. A conventional responding device, such as the controller for a memory bank, monitors the bus for transactions on the fixed subset of addresses that it contains, and responds to relevant read or write operations possibly after some number of "wait" cycles. It may even have to place data on the bus. The cache controller is not responsible for a fixed subset of addresses, but must monitor the bus and perform a tag check on every transaction to determine if the transaction is relevant. For an update-based protocol, many caches may need to snoop the new data off the bus as well. Most modern microprocessors implement such enhanced cache controllers so that they are "multiprocessor-ready".

6.3.2 Reporting snoop results

Snooping introduces a new element to the bus transaction as well. In a conventional bus transaction on a uniprocessor system, one device (the initiator) places an address on the bus, all other devices monitor the address, and one device (the responder) recognizes it as being relevant. Then data is transferred between the two devices. The responder acknowledges its role by raising a

wired-OR signal; if no device decides to respond within a time-out window, a bus error occurs. For snooping caches, each cache must check the address against its tags and the collective result of the snoop from *all* caches must be reported on the bus before the transaction can proceed. In particular, the snoop result informs main memory whether it should respond to the request or some cache is holding a modified copy of the block so an alternative action is necessary. The questions are when the snoop result is reported on the bus, and how.

Let us focus first on the "when" question. Obviously, it is desirable to keep this delay as small as possible so main memory can decide quickly what to do.[1] There are three major options:

1. The design could guarantee that the snoop results are available within a *fixed* number of clock cycles from the issue of the address on the bus. This, in general, requires the use of a dual set of tags because otherwise the processor, which usually has priority, could be accessing the tags heavily when the bus transaction appears. Even with a dual set of tags, one may need to be conservative about the fixed snoop latency, because both sets of tags are made inaccessible when the processor updates the tags; for example in the E --> M state transition in the Illinois MESI protocol.[2] The advantages of this option are a simple memory system design, and disadvantages are extra hardware and potentially longer snoop latency. The Pentium Pro Quads use this approach—with the ability to extend the snoop phase when necessary (see Chapter 8) as do the HP Corporate Business Servers [CAH+93] and the Sun Enterprise.

2. The design could alternatively support a *variable* delay snoop. The main memory assumes that one of the caches will supply the data, until all the cache controllers have snooped and indicated otherwise. This option may be easier to implement since cache controllers do not have to worry about tag-access conflicts inhibiting a timely lookup, and it can offer higher performance since the designer does not have to conservatively assume the worst-case delay for snoop results. The SGI Challenge multiprocessors use a slight variant of this approach, where the memory subsystem goes ahead and fetches the data to service the request, but then stalls if the snoops have not completed by then [GaW93].

3. A third alternative is for the main memory subsystem to maintain a bit per block that indicates whether this block is modified in one of the caches or not. This way the memory subsystem does not have to rely on snooping to decide what action to take, and when controllers return their snoop results is a performance issue. The disadvantage is the extra complexity added to the main-memory subsystem.

How should snoop results be reported on the bus? For the MESI scheme, the requesting cache controller needs to know whether the requested memory block is in other processors' caches, so it can decide whether to load the block in exclusive (E) or shared (S) state. Also, the memory system needs to know whether any cache has the block in modified state, so that memory need not

---

1. Note, that on an atomic bus there are ways to make the system less sensitive to the snoop delay. Since only one memory transaction can be outstanding at any given time, the main memory can start fetching the memory block regardless of whether it or the cache would eventually supply the data; the main-memory subsystem would have sit idle otherwise. Reducing this delay, however, is very important for a split transaction-bus, discussed later. There, multiple bus transactions can be outstanding, so the memory subsystem can be used in the meantime to service another request, for which *it* (and not the cache) may have to supply the data.

2. It is interesting that in the base 3-state invalidation protocol we described, a cache-block state is never updated unless a corresponding bus-transaction is also involved. This usually gives plenty of time to update the tags.

respond. One reasonable option is to use three wired-or signals, two for the snoop results and one indicating that the snoop result is valid. The first signal is asserted when any of the processors' caches (excluding the requesting processor) has a copy of the block. The second is asserted if any processor has the block modified in its cache. We don't need to know the identity of that processor, since it itself knows what action to take. The third signal is an inhibit signal, asserted until all processors have completed their snoop; when it is de-asserted, the requestor and memory can safely examine the other two signals. The full Illinois MESI protocol is more complex because a block can be preferentially retrieved from another cache rather than from memory even if it is in shared state. If multiple caches have a copy, a priority mechanism is needed to decide which cache will supply the data. This is one reason why most commercial machines that use the MESI protocol limit cache-to-cache transfers. The Silicon Graphics Challenge and the Sun Enterprise Server use cache-to-cache transfers only for data that are potentially in modified state in the cache (i.e. are either in exclusive or modified state), in which case there is a single supplier. The Challenge updates memory in the process of a cache to cache transfer, so it does not need a shared-modified state, while the Enterprise does not update memory and uses a shared modified state.

6.3.3 Dealing with Writebacks

Writebacks also complicate implementation, since they involve an incoming block as well as an outgoing one. In general, to allow the processor to continue as soon as possible on a cache miss that causes a writeback (replacement of modified data in the cache), we would like to delay the writeback and instead first service the miss that caused it. This optimization imposes two requirements. First, it requires the machine to provide additional storage, a *writeback buffer*, where the block being replaced can be temporarily stored while the new block is brought into the cache and before the bus can be re-acquired for a second transaction to complete the writeback. Second, before the writeback is completed, it is possible that we see a bus transaction containing the address of that block. In that case, the controller must supply the data from the writeback buffer and cancel its earlier pending request to the bus for a writeback. This requires that an address-comparator be added to snoop on the writeback buffer as well. We will see in Chapter 6 that writebacks introduce further correctness subtleties in machines with physically distributed memory.

6.3.4 Base Organization

Figure 6-4 shows a block-diagram for our resulting base snooping architecture. Each processor has a single-level write-back cache. The cache is dual tagged, so the bus-side controller and the processor-side controller can do tag checks in parallel. The processor-side controller initiates a transaction by placing an address and command on the bus. On a writeback transaction, data is conveyed from the write-back buffer. On a read transaction, it is captured in the data buffer. The bus-side controller snoops the write-back tag as well as the cache tags. Bus arbitration places the requests that go on the bus in a total order. For each transaction, the command and address in the request phase drive the snoop operation—in this total order. The wired-OR snoop results serve as acknowledgment to the initiator that all caches have seen the request and taken relevant action. This defines the operation as being completed, as all subsequent reads and writes are treated as occurring "after" this transaction. It does not matter that the data transfer occurs a few cycles later, because all the caches know that it is coming and can defer servicing additional processor requests until it arrives.

Using this simple design, let us examine more subtle correctness concerns that either require the state machines and protocols to be extended or require care in implementation: non-atomic state transitions, serialization, deadlock, and starvation.
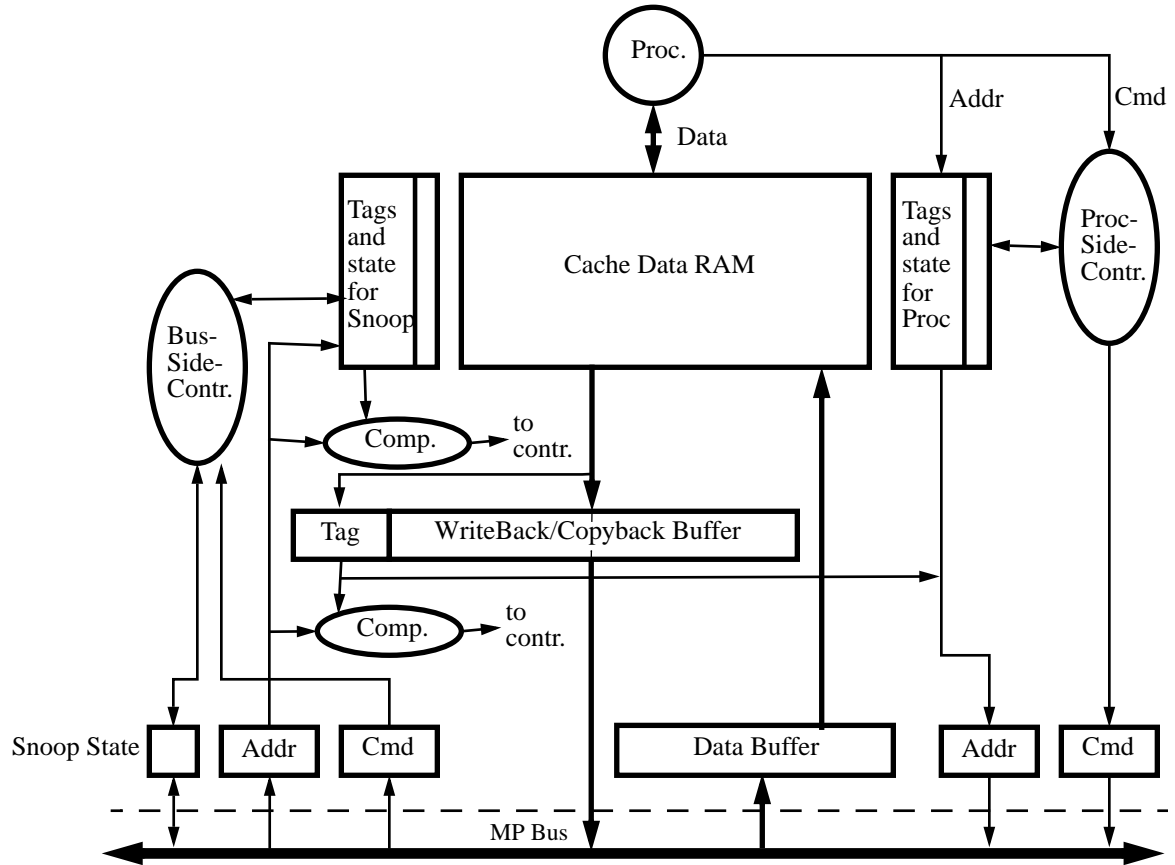
Figure 6-4 Design of a snooping cache for the base machine.

We assume each processor has a single-level writeback cache, an invalidation protocol is used, the processor can have only one memory request outstanding, and that the system bus is atomic. To keep the figure simple, we do not show the bus-arbitration logic and some of the low-level signals and buffers that are needed. We also do not show the coordination signals needed between the bus-side controller and the processor-side controller.

6.3.5 Non-atomic State Transitions

The state transitions and their associated actions in our diagrams have so far been assumed to happen instantaneously or at least atomically. In fact, a request issued by a processor takes some time to complete, often including a bus transaction. While the bus transaction itself is atomic in our simple system, it is only one among the set of actions needed to satisfy a processor's request. These actions include looking up the cache tags, arbitrating for the bus, the actions taken by other controllers at their caches, and the action taken by the issuing processor's controller at the end of the bus transaction (which may include actually writing data into the block). Taken as a whole, the set is not atomic. Even with an atomic bus, multiple requests from different processors may be outstanding in different parts of the system at a time, and it is possible that while a processor

(or controller) P has a request outstanding—for example waiting to obtain bus access—a request from another processor may appear on the bus and need some service from P, perhaps even for the same memory block as P's outstanding request. The types of complications that arise are best illustrated through an example.

Example 6-1

Suppose two processors P1 and P2 cache the same memory block A in shared state, and both simultaneously issue a write to block A. Show how P1 may have a request outstanding waiting for the bus while a transaction from P2 appears on the bus, and how you might solve the complication that results.

Answer

Here is a possible scenario. P1's write will check its cache, determine that it needs to elevate the block's state from shared to modified before it can actually write new data into the block, and issue an upgrade bus request. In the meantime, P2 has also issued a similar upgrade or read-exclusive transaction for A, and it may have won arbitration for the bus first. P1's snoop will see the bus transaction, and must downgrade the state of block A from shared to invalid in its cache. Otherwise when P2's transaction is over A will be in modified state in P2's cache and in shared state in P1's cache, which violates the protocol. To solve this problem, a controller must be able to check addresses snooped from the bus against its own outstanding request, and modify the latter if necessary. Thus, when P1 observes the transaction from P2 and downgrades its block state to invalid, the upgrade bus request it has outstanding is no longer appropriate and must be replaced with a read-exclusive request. (If there were no upgrade transactions in the protocol and read-exclusives were used even on writes to blocks in shared state, the request would not have to be changed in this case even though the block state would have to be changed. These implementation requirements should be considered when assessing the complexity of protocol optimizations.)

A convenient way to deal with the "non-atomic" nature of state transitions, and the consequent need to sometimes revise requests based on observed events, is to expand the protocol state diagram with intermediate or *transient* states. For example, a separate state can be used to indicate that an upgrade request is outstanding. Figure 6-5 shows an expanded state diagram for a MESI protocol. In response to a processor write operation the cache controller begins arbitration for the bus by asserting a bus request, and transitions to the S->M intermediate state. The transition out of this state occurs when the bus arbiter asserts BusGrant for this device. At this point the BusUpgr transaction is placed on the bus and the cache block state is updated. However, if a BusRdX or BusUpgr is observed for this block while in the S->M state, the controller treats its block as having been invalidated before the transaction. (We could instead retract the bus request and transition to the I state, whereupon the still pending PrWr would be detected again.) On a processor read from invalid state, the controller advances to an intermediate state (I-->S,E); the next stable state to transition to is determined by the value of the shared line when the read is granted the bus. These intermediate states are typically not encoded in the cache block state bits, which are still MESI, since it would be wasteful to expend bits in every cache slot to indicate the one block in the cache that may be in a transient state. They are reflected in the combination of state bits and controller state. However, when we consider caches that allow multiple outstanding transactions, it will be necessary to have an explicit representation for the (multiple) blocks from a cache that may be in a transient state.
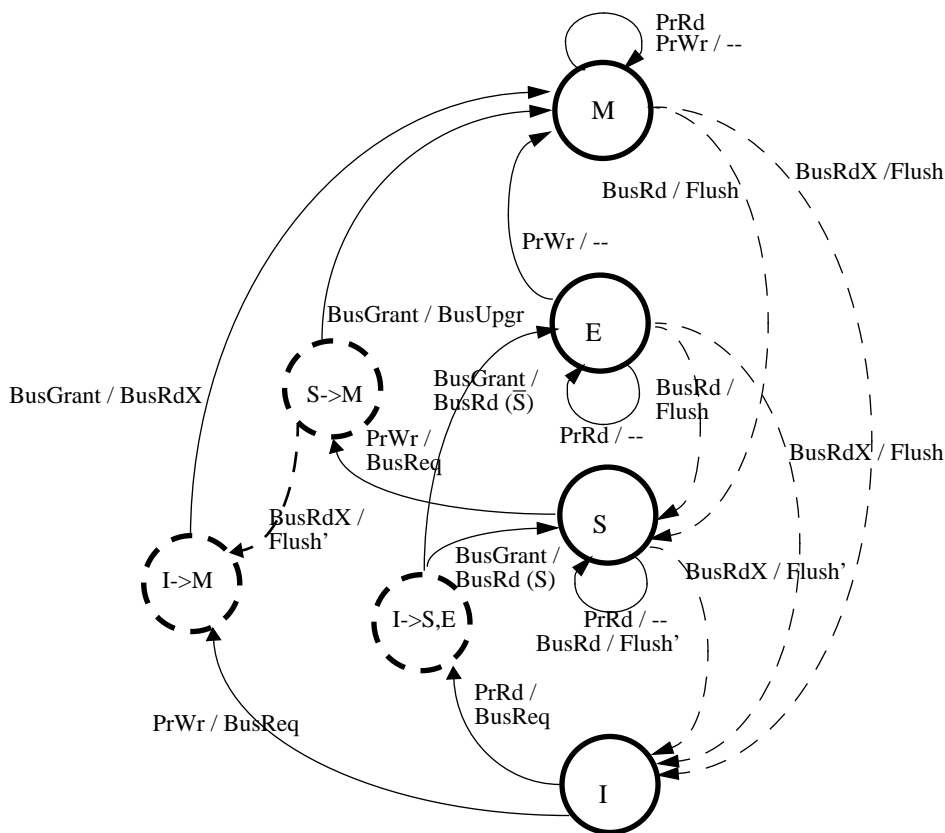
Figure 6-5 Expanded MESI protocol state diagram indicating transient states for bus acquisition

Expanding the number of states in the protocol greatly increases the difficulty of proving that an implementation is correct, or of testing the design. Thus, designers seek mechanisms that avoid transient states. The Sun Enterprise, for example, does not use a BusUpgr transaction in the MESI protocol, but uses the result of the snoop to eliminate unnecessary data transfers in the BusRdX. Recall, that on a BusRdX the caches holding the block invalidate their copy. If a cache has the block in the dirty state, it raises the dirty line, thereby preventing the memory from supplying the data, and flushes the data onto the bus. No use is made of the shared line. The trick is to have the processor that issues the BusRdX to snoop its own tags when the transaction actually goes on the bus. If the block is still in its cache, it raises the shared line, which waves off the memory. Since it already has the block, the data phase of the transaction is skipped. The cache controller does not need a transient state, because regardless of what happens it has one action to take—place a BusRdX transaction on the bus.

6.3.6 Serialization

With the non-atomicity of an entire memory operation, care must be taken in the processor-cache handshake to preserve the order determined by the serialization of bus transactions. For a read, the processor needs the result of the operation. To gain greater performance on writes, it is tempt-

ing to update the cache block and allow the processor continue with useful instructions while the cache controller acquires exclusive ownership of the block—and possibly loads the rest of the block—via a bus transaction. The problem is that there is a window between the time the processor gives the write to the cache and when the cache controller acquires the bus for the read-exclusive (or upgrade) transaction. Other bus transactions may occur in this window, which may change the state of this or other blocks in the cache. To provide write serialization and SC, these transactions must appear to the processor as occurring before the write, since that is how they are serialized by the bus and appear to other processors. Conservatively, the cache controller should not allow the processor issuing the write to proceed past the write until the read-exclusive transaction occurs on the bus and makes the write visible to other processors. In particular, even for write serialization the issuing processor should not consider the write complete until it has appeared on the bus and thus been serialized with respect to other writes.

In fact, the cache does not have to wait until the read-exclusive transaction is finished—i.e. other copies have actually been invalidated in their caches—before allowing the processor to continue; it can even service read and write hits once the transaction is on the bus, assuming access to the block in transit is handled properly. The crux of the argument for coherence and for sequential consistency presented in Section 5.4 was that all cache controllers observe the exclusive ownership transactions (generated by write operations) in the same order and that the write occurs immediately after the exclusive ownership transaction. Once the bus transaction starts, the writer knows that all other caches will invalidate their copies before another bus transaction occurs. The position of the write in the serial bus order is *committed*. The writer never knows where exactly the invalidation is inserted in the local program order of the other processors; it knows only that it is before whatever operation generates the next bus transaction and that all processors insert the invalidations in the same order. Similarly, the writer's local sequence of memory operations only become visible at the next bus transaction. This is what is important to maintain the necessary orderings for coherence and SC, and it allows the writer to *substitute commitment for actual completion* in following the sufficient conditions for SC. In fact, it is the basic observation that makes it possible to implement cache coherency and sequential consistency with pipelined busses, multilevel memory hierarchies, and write buffers. Write atomicity follows the same argument as presented before in Section 5.4.

This discussion of serialization raises an important, but somewhat subtle point. Write serialization and write atomicity have very little to do with when the write-backs occur or with when the actual location in memory is updated. Either a write or a read can cause a write-back, if it causes a dirty block to be replaced. The write-backs are bus transactions, but they do not need to be ordered. On the other hand, a write does not necessarily cause a write-back, even if it misses; it causes a read-exclusive. What is important to the program is when the new value is bound to the address. The write completes, in the sense that any subsequent read will return the new or later value once the BusRdX or BusUpgr transaction takes place. By invalidating the old cache blocks, it ensures that all reads of the old value precede the transaction. The controller issuing the transaction ensures that the write occurs after the bus transaction, and that no other memory operations intervene.

6.3.7 Deadlock

A two-phase protocol, such as the request-response protocol of a memory operation, presents a form of protocol-level deadlock, sometimes called *fetch-deadlock*[Lei*92], that is not simply a question of buffer usage. While an entity is attempting to issue its request, it needs to service

incoming transactions. The place where this arises in an atomic bus based SMP, is that while the cache controller is awaiting the bus grant, it needs to continue performing snoops and handling requests which may cause it to flush blocks onto the bus. Otherwise, the system may deadlock if two controllers have outstanding transactions that need to be responded to by each other, and both are refusing to handle requests. For example, suppose a BusRd for a block B appears on the bus while a processor P1 has a read-exclusive request outstanding to another block A and is waiting for the bus. If P1 has a modified copy of B, its controller should be able to supply the data and change the state from modified to shared while it is waiting to acquire the bus. Otherwise the current bus transaction is waiting for P1's controller, while P1's controller is waiting for the bus transaction to release the bus.

6.3.8 Livelock and Starvation

The classic potential livelock problem in an invalidation-based cache-coherent memory system is caused by all processors attempting to write to the same memory location. Suppose that initially no processor has a copy of the location in its cache. A processor's write requires the following non-atomic set of events: Its cache obtains exclusive ownership for the corresponding memory block—i.e. invalidates other copies and obtains the block in modified state—a state machine in the processor realizes that the block is now present in the cache in the appropriate state, and the state-machine re-attempts the write. Unless the processor-cache handshake is designed carefully, it is possible that the block is brought into the cache in modified state, but before the processor is able to complete its write the block is invalidated by a BusRdX request from another processor. The processor misses again and this cycle can repeat indefinitely. To avoid livelock, a write that has obtained exclusive ownership must be allowed to complete before the exclusive ownership is taken away.

With multiple processors competing for a bus, it is possible that some processors may be granted the bus repeatedly while others may not and may become starved. Starvation can be avoided by using first-come-first-serve service policies at the bus and elsewhere. These usually require additional buffering, so sometimes heuristic techniques are used to reduce the likelihood of starvation instead. For example, a count can be maintained of the number of times that a request has been denied and, after a certain threshold, action is taken so that no other new request is serviced until this request is serviced.

6.3.9 Implementing Atomic Primitives

The atomic operations that the above locks need, test&set and fetch&increment, can be implemented either as individual atomic read-modify-write instructions or using an LL-SC pair. Let us discuss some implementation issues for these primitives, before we discuss all-hardware implementations of locks. Let us first consider the implementation of atomic exchange or read-modify-write instructions first, and then LL-SC.

Consider a simple test&set instruction. It has a read component (the test) and a write component (the set). The first question is whether the test&set (lock) variable should be cacheable so the test&set can be performed in the processor cache, or it should be uncacheable so the atomic operation is performed at main memory. The discussion above has assumed cacheable lock variables. This has the advantage of allowing locality to be exploited and hence reducing latency and traffic when the lock is repeatedly acquired by the same processor: The lock variable remains dirty in the cache and no invalidations or misses are generated. It also allows processors to spin in their

caches, thus reducing useless bus traffic when the lock is not ready. However, performing the operations at memory can cause faster transfer of a lock from one processor to another. With cacheable locks, the processor that is busy-waiting will first be invalidated, at which point it will try to access the lock from the other processor's cache or memory. With uncached locks the release goes only to memory, and by the time it gets there the next attempt by the waiting processor is likely to be on its way to memory already, so it will obtain the lock from memory with low latency. Overall, traffic and locality considerations tend to dominate, and lock variables are usually cacheable so processors can busy-wait without loading the bus.

A conceptually natural way to implement a cacheable test&set that is not satisfied in the cache itself is with two bus transactions: a read transaction for the test component and a write transaction for the set component. One strategy to keep this sequence atomic is to lock down the bus at the read transaction until the write completes, keeping other processors from putting accesses (especially to that variable) on the bus between the read and write components. While this can be done quite easily with an atomic bus, it is much more difficult with a split-transaction bus: Not only does locking down the bus impact performance substantially, but it can cause deadlock if one of the transactions cannot be immediately satisfied without giving up the bus.

Fortunately, there are better approaches. Consider an invalidation-based protocol with write-back caches. What a processor really needs to do is obtain exclusive ownership of the cache block (e.g. by issuing a single read-exclusive bus transaction), and then it can perform the read component and the write component in the cache as long as it does not give up exclusive ownership of the block in between, i.e. it simply buffers and delays other incoming accesses from the bus to that block until the write component is performed. More complex atomic operations such as fetch-and-op must retain exclusive ownership until the operation is completed.

An atomic instruction that is more complex to implement is compare-and-swap. This requires specifying three operands in a memory instruction: the memory location, the register to compare with, and the value/register to be swapped in. RISC instruction sets are usually not equipped for this.

Implementing LL-SC requires a little special support. A typical implementation uses a so called lock-flag and a lock-address register at each processor. An LL operation reads the block, but also sets the lock-flag and puts the address of the block in the lock-address register. Incoming invalidation (or update) requests from the bus are matched against the lock-address register, and a successful match (a conflicting write) resets the lock flag. An SC checks the lock flag as the indicator for whether an intervening conflicting write has occurred; if so it fails, and if not it succeeds. The lock flag is also reset (and the SC will fail) if the lock variable is replaced from the cache, since then the processor may no longer see invalidations or updates to that variable. Finally, the lock-flag is reset at context switches, since a context switch between an LL and its SC may incorrectly cause the LL of the old process to lead to the success of an SC in the new process that is switched in.

Some subtle issues arise in avoiding livelock when implementing LL-SC. First, we should in fact not allow replacement of the cache block that holds the lock variable between the LL and the SC. Replacement would clear the lock flag, as discussed above, and could establish a situation where a processor keeps trying the SC but never succeeds due to continual replacement of the block between LL and SC. To disallow replacements due to conflicts with instruction fetches, we can use split instruction and data caches or set-associative unified caches. For conflicts with other data references, a common solution is to simply disallow memory instructions between an LL

and an SC. Techniques to hide latency (e.g. out-of-order issue) can complicate matters, since memory operations that are not between the LL and SC in the program code may be so in the execution. A simple solution to this problem is to not allow reorderings past LL or SC operations.

The second potential livelock situation would occur if two processes continually fail on their SCs, and each process's failing SC invalidated or updated the other process's block thus clearing the lock-flag. Neither of the two processes would ever succeed if this pathological situation persisted. This is why it is important that an SC not be treated as an ordinary write, but that it not issuing invalidations or updates when it fails.

Note that compared to implementing an atomic read-modify-write instruction, LL-SC can have a performance disadvantage since both the LL and the SC can miss in the cache, leading to two misses instead of one. For better performance, it may be desirable to obtain (or prefetch) the block in exclusive or modified state at the LL, so the SC does not miss unless it fails. However, this reintroduces the second livelock situation above: Other copies are invalidated to obtain exclusive ownership, so their SCs may fail without guarantee of this processor's SC succeeding. If this optimization is used, some form of backoff should be used between failed operations to minimize (though not completely eliminate) the probability of livelock.

6.4 Multi-level Cache Hierarchies

The simple design above was illustrative, but it made two simplifying assumptions that are not valid on most modern systems: It assumed single-level caches, and an atomic bus. This section relaxes the first assumption and examines the resulting design issues.

The trend in microprocessor design since the early 90's is to have an on-chip first-level cache and a much larger second-level cache, either on-chip or off-chip.[1] Some systems, like the DEC Alpha, use on-chip secondary caches as well and an off-chip tertiary cache. Multilevel cache hierarchies would seem to complicate coherence since changes made by the processor to the first-level cache may not be visible to the second-level cache controller responsible for bus operations, and bus transactions are not directly visible to the first level cache. However, the basic mechanisms for cache coherence extend naturally to multi-level cache hierarchies. Let us consider a two-level hierarchy for concreteness; the extension to the multi-level case is straightforward.

One obvious way to handle multi-level caches is to have independent bus snooping hardware for each level of the cache hierarchy. This is unattractive for several reasons. First, the first-level cache is usually on the processor chip, and an on-chip snooper will consume precious pins to monitor the addresses on the shared bus. Second, duplicating the tags to allow concurrent access by the snooper and the processor may consume too much precious on-chip real estate. Third, and more importantly, there is duplication of effort between the second-level and first-level snoops, since most of the time blocks present in the first-level cache are also present in the second-level cache so the snoop of the first-level cache is unnecessary.

---

1. The HP PA-RISC microprocessors are a notable exception, maintaining a large off-chip first level cache for many years after other vendors went to small on-chip first level caches.

The solution used in practice is based on this last observation. When using multi-level caches, designers ensure that they preserve the inclusion property. The *inclusion property* requires that:

1. If a memory block is in the first-level cache, then it must also be present in the second-level cache. In other words, the contents of the first-level cache must be a subset of the contents of the second-level cache.

2. If the block is in a modified state (e.g., modified or shared-modified) in the first-level cache, then it must also be marked modified in the second-level cache.

The first requirement ensures that all bus transactions that are relevant to the L1 cache are also relevant to L2 cache, so having the L2 cache controller snoop the bus is sufficient. The second ensures that if a BusRd transaction requests a block that is in modified state in the L1 or L2 cache, then the L2 snoop can immediately respond to the bus.
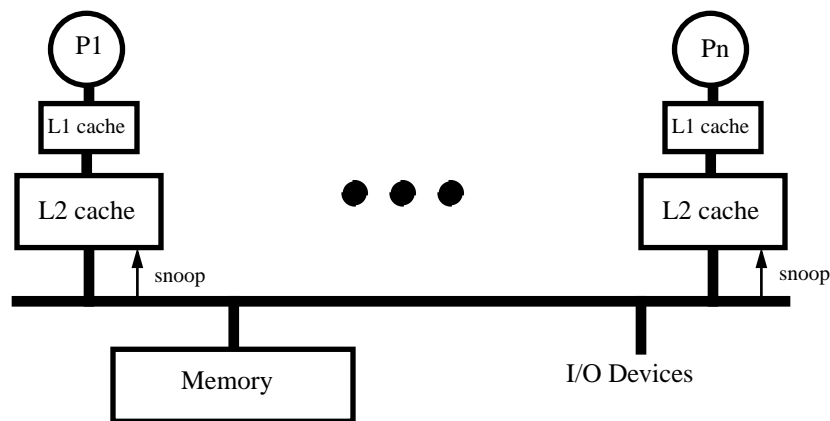
Figure 6-6 A bus-based machine containing processors with multi-level caches.

Coherence is maintained by ensuring the inclusion property, whereby all blocks in the L1 cache are also present in the L2 cache. With the inclusion property, a L2-snoop also suffices for the L1 cache.

6.4.1 Maintaining inclusion

The requirements for inclusion are not trivial to maintain. Three aspects need to be considered. Processor references to the first level cache cause it to change state and perform replacements; these need to be handled in a manner that maintains inclusion. Bus transactions cause the second level cache to change stage and flush blocks; these need to be forwarded to the first level. Finally, the modified state must be propagated out to the second level cache.

At first glance it might appear that inclusion would be satisfied automatically, since all first level cache misses go to the second level cache. The problem, however, is that two caches may choose different blocks to replace. Inclusion falls out automatically only for certain combinations of cache configuration. It is an interesting exercise to see what conditions can cause inclusion to be violated if no special care is taken, so let us do that a little here before we look at how inclusion is typically maintained. Consider some examples of typical cache hierarchies [BaW88]. For notational purposes, assume that the L1 cache has associativity a_1, number of sets n_1, block size b_1, and thus a total capacity of $S_1 = a_1.b_1.n_1$, where a_1, b_1 and n_1 are all powers of two. The corre-

sponding parameters for the L2 cache are a_2, n_2, b_2, and S_2. We also assume that all parameter values are powers of 2.

Set-associative L1 caches with history-based replacement. The problem with replacement polices based on the history of accesses to a block, such as least recently used (LRU) replacement, is that the L1 cache sees a different history of accesses than L2 and other caches, since all processor references look up the L1 cache but not all get to lower-level caches. Suppose the L1 cache is two-way set-associative with LRU replacement, both L1 and L2 caches have the same block size ($b_1 = b_2$), and L2 is k times larger than L1 ($n_2 = k.n_1$). It is easy to show that inclusion does *not* hold in this simple case. Consider three distinct memory blocks m_1, m_2, and m_3 that map to the same set in the L1 cache. Assume that m_1 and m_2 are currently in the two available slots within that set in the L1 cache, and are present in the L2 cache as well. Now consider what happens when the processor references m_3, which happens to collide with and replace one of m_1 and m_2 in the L2 cache as well. Since the L2 cache is oblivious of the L1 cache's access history which determines whether the latter replaces m_1 or m_2, it is easy to see that the L2 cache may replace one of m_1 and m_2 while the L1 cache may replace the other. This is true even if the L2 cache is two-way set associative and $m1$ and $m2$ fall into the same set in it as well. In fact, we can generalize the above example to see that inclusion can be violated if L1 is not direct-mapped and uses an LRU replacement policy, regardless of the associativity, block size, or cache size of the L2 cache.

Multiple caches at a level. A similar problem with replacements is observed when the first level caches are split between instructions and data, even if they are direct mapped, and are backed up by a unified second-level cache. Suppose first that the L2 cache is direct mapped as well. An instruction block m1 and a data block m2 that conflict in the L2 cache do not conflict in the L1 caches since they go into different caches. If m2 resides in the L2 cache and m1 is referenced, m2 will be replaced from the L2 cache but not from the L1 data cache, violating inclusion. Set associativity in the unified L2 cache doesn't solve the problem. For example, suppose the L1 caches are direct-mapped and the L2 cache is 2-way set associative. Consider three distinct memory blocks m_1, m_2, and m_3, where m_1 is an instruction block, m_2 and m_3 are data blocks that map to the same set in the L1 data cache, and all three map to a single set in the L2 cache. Assume that at a given time m_1 is in the L1 instruction cache, m_2 is in the L1 data cache, and they occupy the two slots within a set in the L2 cache. Now when the processor references m_3, it will replace m_2 in the L1 data cache, but the L2 cache may decide to replace m_1 or m_2 depending on the past history of accesses to it, thus violating inclusion. This can be generalized to show that if there are multiple independent caches that connect to even a highly associative cache below, inclusion is not guaranteed.

Different cache block sizes. Finally, consider caches with different block sizes. Consider a miniature system with a direct-mapped, unified L1 and L2 caches ($a_1 = a_2 = 1$), with block sizes 1-word and 2-words respectively ($b_1 = 1$, $b_2 = 2$), and number of sets 4 and 8 respectively ($n_1 = 4$, $n_2 = 8$). Thus, the size of L1 is 4 words, and locations 0, 4, 8,... map to set 0, locations 1, 5, 9,... map to set 1, and so on. The size of L2 is 16 words, and locations 0&1, 16&17, 32&33,... map to set-0, locations 2&3, 18&19, 34&35,... map to set-1, and so on. It is now easy to see that while the L1 cache can contain both locations 0 and 17 at the same time (they map to sets 0 and 1 respectively), the L2 cache can not do so because they map to the same set (set-0) and they are not consecutive locations (so block size of 2-words does not help). Hence inclusion can be violated. Inclusion can be shown to be violated even if the L2 cache is much larger or has greater associativity, and we have already seen the problems when the L1 cache has greater associativity.

In one of the most commonly encountered cases, inclusion *is* maintained automatically. This is when the L1 cache is direct-mapped ($a_1 = 1$), L2 can be direct-mapped or set associative ($a_2 >= 1$) with any replacement policy (e.g., LRU, FIFO, random) as long as the new block brought in is put in both L1 and L2 caches, the block-size is the same ($b_1 = b_2$), and the number of sets in the L1 cache is equal to or smaller than in L2 cache ($n_1 =< n_2$). Arranging this configuration is one popular way to get around the inclusion problem.

However, many of the cache configurations used in practice do not automatically maintain inclusion on replacements. Instead, the mechanism used for propagating coherency events is extended to explicitly maintain inclusion. Whenever a block in the L2 cache is replaced, the address of that block is sent to the L1 cache, asking it to invalidate or flush (if dirty) the corresponding blocks (there can be multiple blocks if $b_2 > b_1$).

Now consider bus transactions seen by the L2 cache. Some, but not all, of the bus transactions relevant to the L2 cache are also relevant to the L1 cache and must be propagated to it. For example, if a block is invalidated in the L2 cache due to an observed bus transaction (e.g., BusRdX), the invalidation must also be propagated to the L1 cache if the data are present in it. There are several ways to do this. One is to inform the L1 cache of all transactions that were relevant to the L2 cache, and to let it ignore the ones whose addresses do not match any of its tags. This sends a large number of unnecessary interventions to the L1 cache and can hurt performance by making cache-tags unavailable for processor accesses. A more attractive solution is for the L2 cache to keep extra state (inclusion bits) with cache blocks, which record whether the block is also present in the L1 cache. It can then suitably filter interventions to the L1 cache, at the cost of slightly extra hardware and complexity.

Finally, on a L1 write hit, the modification needs to be communicated to the L2 cache. One solution is to make the L1 cache write-through. This has the advantage that single-cycle writes are simple to implement [HP95]. However, writes can consume a substantial fraction of the L2 cache bandwidth, and a write-buffer is needed between the L1 and L2 caches to avoid processor stalls. The requirement can also be satisfied with writeback L1 caches, since it is not necessary that the data in the L2 cache be up-to-date but only that the L2 cache knows when the L1 cache has more recent data. Thus, the state of the L2 cache blocks is augmented so that blocks can be marked "*modified-but-stale.*" The block behaves as modified for the coherency protocol, but data is fetched from the L1 cache on a flush. (One simple approach is to set both the modified and invalid bits.) Both the write-through and writeback solutions have been used in many bus-based multiprocessors. More information on maintaining cache inclusion can be found in [BaW88].

6.4.2 Propagating transactions for coherence in the hierarchy

Given that we have inclusion and we propagate invalidations and flushes up to the L1 cache as necessary, let us see how transactions percolate up and down a processor's cache hierarchy. The intra-hierarchy protocol handles processor requests by percolating them downwards (away from the processor) until either they encounter a cache which has the requested block in the proper state (shared or modified for a read request, and modified for a write/read-exclusive request) or they reach the bus. Responses to these processor requests are sent up the cache hierarchy, updating each cache as they progress towards the processor. Shared responses are loaded into each cache in the hierarchy in the shared state, while read-exclusive responses are loaded into all levels, except the innermost (L1) in the modified-but-stale state. In the innermost cache, read-exclusive data are loaded in the modified state, as this will be the most up-to-date copy.

External flush, copyback, and invalidate requests from the bus percolate upward from the external interface (the bus), modifying the state of the cache blocks as they progress (changing the state to invalid for flush and copyback requests, and to shared for copyback requests). Flush and copyback requests percolate upwards until they encounter the modified copy, at which point a response is generated for the external interface. For invalidations, it is not necessary for the bus transaction to be held up until all the copies are actually invalidated. The lowest-level cache controller (closest to the bus) sees the transaction when it appears on the bus, and this serves as a point of commitment to the requestor that the invalidation will be performed in the appropriate order. The response to the invalidation may be sent to the processor from its own bus interface as soon as the invalidation request is placed on the bus, so no responses are generated within the destination cache hierarchies. All that is required is that certain orders be maintained between the incoming invalidations and other transactions flowing through the cache hierarchy, which we shall discuss further in the context of split transaction busses that allow many transactions to be outstanding at a time.

Interestingly, dual tags are less critical when we have multi-level caches. The L2 cache acts as a filter for the L1 cache, screening out irrelevant transactions from the bus, so the tags of the L1 cache are available almost wholly to the processor. Similarly, since the L1 cache acts as a filter for the L2 cache from the processor side (hopefully satisfying most of processor's requests), the L2 tags are almost wholly available for bus snooper's queries (see Figure 6-7). Nonetheless, many machines retain dual tags even in multilevel cache designs.
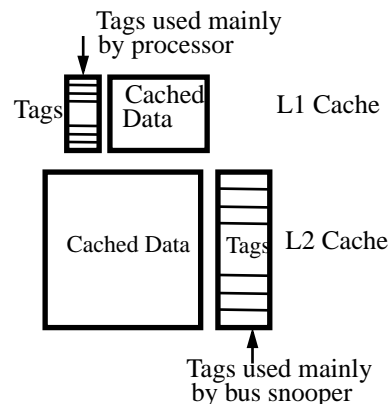
Figure 6-7 Organization of two-level snoopy caches.

With only one outstanding transaction on the bus at a time, the major correctness issues do not change much by using a multi-level hierarchy as long as inclusion is maintained. The necessary transactions are propagated up and down the hierarchy, and bus transactions may be held up until the necessary propagation occurs. Of course, the performance penalty for holding the processor write until the BusRdX has been granted is more onerous, so we are motivated to try to decouple these operations (the penalty for holding the bus until the L1 cache is queried is avoided by the early commitment optimization above). Before going further down this path, let us remove the second simplifying assumption, of an atomic bus, and examine a more aggressive, split-transaction bus. We will first return to assuming a single-level processor cache for simplicity, and then incorporate multi-level cache hierarchies.

6.5 Split-transaction Bus

An atomic bus limits the achievable bus bandwidth substantially, since the bus wires are idle for the duration between when the address is sent on the bus and when the memory system or another cache supply the data or response. In a split-transaction bus, transactions that require a response are split into two independent sub-transactions, a *request* transaction and a *response* transaction. Other transactions (or sub-transactions) are allowed to intervene between them, so that the bus can be used while the response to the original request is being generated. Buffering is used between the bus and the cache controllers to allow multiple transactions to be outstanding on the bus waiting for snoop and/or data responses from the controllers. The advantage, of course, is that by pipelining bus operations the bus is utilized more effectively and more processors can share the same bus. The disadvantage is increased complexity.

As examples of request-response pairs, a BusRd transaction is now a request that needs a data response. A BusUpgr does not need a data response, but it does require an acknowledgment indicating that it has committed and hence been serialized. This acknowledgment is usually sent down toward the processor by the requesting processor's bus controller when it is granted the bus for the BusUpgr request, so it does not appear on the bus as a separate transaction. A BusRdX needs a data response and an acknowledgment of commitment; typically, these are combined as part of the data response. Finally, a writeback usually does not have a response.

The major complications caused by split transaction buses are:

1. A new request can appear on the bus before the snoop and/or servicing of an earlier request are complete. In particular, *conflicting* requests (two requests to the same memory block, at least one of which is due to a write operation) may be outstanding on the bus at the same time, a case which must be handled very carefully. This is different from the earlier case of non-atomicity of overall actions using an atomic bus; there, a conflicting request could be observed by a processor but before its request even obtained the bus, so the request could be suitably modified before being placed on the bus.

2. The number of buffers for incoming requests and potentially data responses from bus to cache controller is usually fixed and small, so we must either avoid or handle buffers filling up. This is called *flow control*, since it affects the flow of transactions through the system.

3. Since bus/snoop requests are buffered, we need to revisit the issue of when and how snoop responses and data responses are produced on the bus. For example, are they generated in order with respect to the requests appearing on the bus or not, and are the snoop and the data part of the same response transaction?

Example 6-2   Consider the previous example of two processors P1 and P2 having the block cached in shared state and deciding to write it at the same time (Example 6-1). Show how a split-transaction bus may introduce complications which would not arise with an atomic bus.

Answer   With a split-transaction bus, P1 and P2 may generate BusUpgr requests that are granted the bus on successive cycles. For example, P2 may get the bus before it has been able to look up the cache for P1's request and detect it to be conflicting. If they both assume that they have acquired exclusive ownership, the protocol breaks down

because both P1 and P2 now think they have the block in modified state. On an atomic bus this would never happen because the first BusUpgr transaction would complete—snoops, responses, and all—before the second one got on the bus, and the latter would have been forced to change its request from BusUpgr to BusRdX. (Note that even the breakdown on the atomic bus discussed in Example 6-1 resulted in only one processor having the block in modified state, and the other having it in shared state.)

The design space for split-transaction, cache-coherent busses is large, and a great deal of innovation is on-going in the industry. Since bus operations are pipelined, one high level design decision is whether responses need to be kept in the same order as the requests. The Intel Pentium Pro and DEC Turbo Laser busses are examples of the "in order" approach, while the SGI Challenge and Sun Enterprise busses allow responses to be out of order. The latter approach is more tolerant of variations in memory access times (memory may be able to satisfy a later request quicker than an earlier one, due to memory bank conflicts or off-page DRAM access), but is more complex. A second key decision is how many outstanding requests are permitted: few or many. In general, a larger number of outstanding requests allows better bus utilization, but requires more buffering and design complexity. Perhaps the most critical issue from the viewpoint of the cache coherence protocol is how ordering is established and when snoop results are reported. Are they part of the request phase or the response phase? The position adopted on this issue determines how conflicting operations can be handled. Let us first examine one concrete design fully, and then discuss alternatives.

6.5.1 An Example Split-transaction Design

The example is based loosely on the Silicon Graphics Challenge bus architecture, the PowerPath 2. It takes the following positions on the three issues. Conflicting requests are dealt with very simply, if conservatively: the design disallows multiple requests for a block from being outstanding on the bus at once. In fact, it allows only eight outstanding requests at a time on the bus, thus making the conflict detection tractable. Limited buffering is provided between the bus and the cache controllers, and flow-control for these buffers is implemented through *negative acknowledgment* or *NACK* lines on the bus. That is, if a buffer is full when a request is observed, which can be detected as soon as the request appears on the bus, the request is rejected and NACKed; this renders the request invalid, and asks the requestor to retry. Finally, responses are allowed to be provided in a different order than that in which the original requests appeared on the bus. The request phase establishes the total order on coherence transactions, however, snoop results from the cache controllers are presented on the bus as part of the response phase, together with the data if any.

Let us examine the high-level bus design and how responses are matched up with requests. Then, we shall look at the flow control and snoop result issues in more depth. Finally, we shall examine the path of a request through the system, including how conflicting requests are kept from being simultaneously outstanding on the bus.

6.5.2 Bus Design and Request-response Matching

The split-transaction bus design essentially consists of two separate busses, a request bus for command and address and a response bus for data. The request bus provides the type of request (e.g., BusRd, BusWB) and the target address. Since responses may arrive out of order with regard

to requests, there needs to be a way to identify returning responses with their outstanding requests. When a request (command-address pair) is granted the bus by the arbiter, it is also assigned a unique tag (3-bits, since there are eight outstanding requests in the base design). A response consists of data on the data bus as well as the original request tags. The use of tags means that responses do not need to use the address lines, keeping them available for other requests. The address and the data buses can be arbitrated for separately. There are separate bus lines for arbitration, as well as for flow control and snoop results.

Cache blocks are 128 bytes (1024 bits) and the data bus is 256 bits wide in this particular design, so four cycles plus a one cycle 'turn-around' time are required for the response phase. A uniform pipeline strategy is followed, so the request phase is also five cycles: arbitration, resolution, address, decode, and acknowledgment. Overall, a complete request-response transaction takes three or more of these five-cycle phases, as we shall see. This basic pipelining strategy underlies several of the higher level design decisions. To understand this strategy, let's follow a single read operation through to completion, shown in Figure 6-8. In the request arbitration cycle a cache controller presents its request for the bus. In the request resolution cycle all requests are considered, a single one is granted, and a tag is assigned. The winner drives the address in the following address cycle and then all controllers have a cycle to decode it and look up the cache tags to determine whether there is a snoop hit (the snoop result will be presented on the bus later). At this point, cache controllers can take the action which makes the operation 'visible' to the processor. On a BusRd, an exclusive block is downgraded to shared; on a BusRdX or BusUpgr blocks are invalidated. In either case, a controller owning the block as dirty knows that it will need to flush the block in the response phase. If a cache controller was not able to take the required action during the Address phase, say if it was unable to gain access to the cache tags, it can inhibit the completion of the bus transaction in the A-ack cycle. (During the ack cycle, the first data transfer cycle for the previous data arbitration cycle can take place, occupying the data lines for four cycles, see Figure 6-8.)

After the address phase it is determined which module should respond with the data: the memory or a cache. The responder may request the data bus during a following arbitration cycle. (Note that in this cycle a requestor also initiates a new request on the address bus). The data bus arbitration is resolved in the next cycle and in the address cycle the tag can be checked. If the target is ready, the data transfer starts on the ack cycle and continues for three additional cycles. After a single turn-around cycle, the next data transfer (whose arbitration was proceeding in parallel) can start. The cache block sharing state (snoop result) is conveyed with the response phase and state bits are set when the data is updated in the cache.

As discussed earlier, writebacks (BusWB) consist only of a request phase. They require use of both the address and data lines together, and thus must arbitrate for simultaneous use of both resources. Finally, upgrades (BusUpgr) performed to acquire exclusive ownership for a block also have only a request part since no data response is needed on the bus. The processor performing a write that generates the BusUpgr is sent a response by its own bus controller when the BusUpgr is actually placed on the bus, indicating that the write is committed and has been serialized in the bus order.

To keep track of the outstanding requests on the bus, each cache controller maintains an eight-entry buffer, called a *request table* (see Figure 6-9), Whenever a new request is issued on the bus, it is added to all request tables at the same index, which is the three-bit tag assigned to that request, as part of the arbitration process. A request table entry contains the block address associated with the request, the request type, the state of the block in the local cache (if it has already

Arb	Rslv	Addr	Dcd	Ack	Arb	Rslv	Addr	Dcd	Ack	Arb	Rslv	Addr	Dcd	Ack

Addr Req | Grant | Addr | | A-ack | *Addr Req* | | *Addr* | | *A-ack* |

Data Req | | *D-res* |

Data Req | | *D-res*

D0 | D1 | D2 | D3 | | *D0*
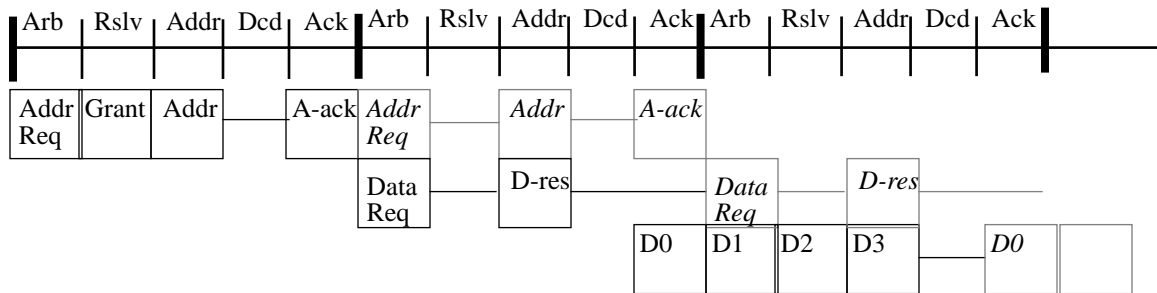
Figure 6-8 Complete read transaction for a split-transaction bus

A pair of consecutive read operations are performed on consecutive phases. Each phase consist of five specific cycles: arbitration, resolution, address, decode, and acknowledgment. Transactions are split into to phases: address and data.

been determined), and a few other bits. The request table is fully associative, so a new request can be physically placed anywhere in the table; all request table entries are examined for a match by both requests issued by this processor and by other requests and responses observed from the bus. A request table entry is freed when a response to the request is observed on the bus. The tag value associated with that request is reassigned by the bus arbiter only at this point, so there are no conflicts in the request tables.

6.5.3 Snoop Results and Conflicting Requests

Like the SGI Challenge, this base design uses variable delay snooping. The snoop portion of the bus consists of the three wired-or lines discussed earlier: *sharing*, *dirty*, and *inhibit*, which extends the duration of the current response sub-transaction. At the end of the request phase, it is determined which module is to respond with the data. However, it may be many cycles before that data is ready and the responder gains access to the data bus. During this time, other requests and responses may take place. The snoop results in this design are presented on the bus by all controllers at the time they see the actual response to a request being put on the bus, i.e. during the response phase. Writeback and upgrade requests do not have a data response, but then they do not require a snoop response either.

Avoiding conflicting requests is easy: Since every controller has a record of the pending reads in its request table, no request is issued for a block that has a response outstanding. Writes are performed during the request phase, so even though the bus is pipelined the operations for an individual location are serialized as in the atomic case. However, this alone does not ensure sequential consistency.

6.5.4 Flow Control

In addition to flow control for incoming requests from the bus, flow control may also be required in other parts of the system. The cache subsystem has a buffer in which responses to its requests can be stored, in addition to the writeback buffer discussed earlier. If the processor or cache allows only one outstanding request at a time, this response buffer is only one entry deep. The number of buffer entries is usually kept small anyway, since a response buffer entry contains not
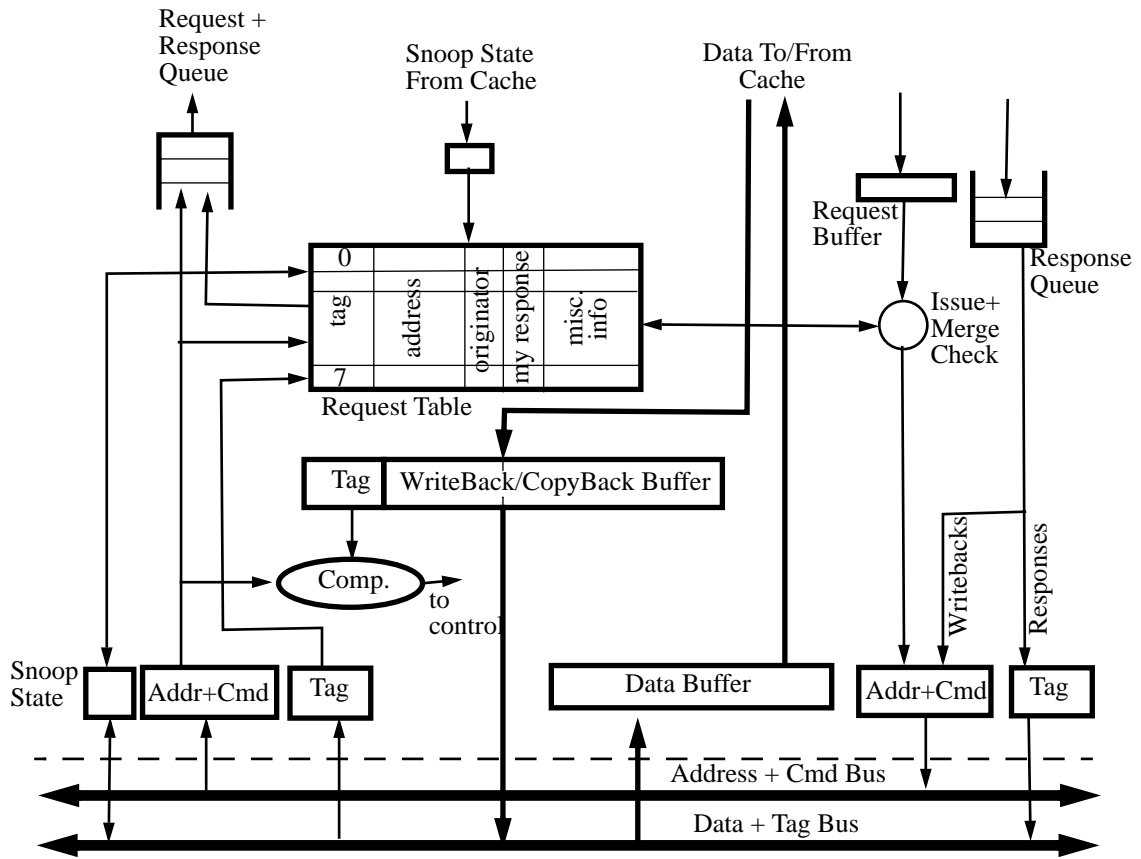
Figure 6-9 Extension of the bus-interface logic shown in Figure 6-4 to accommodate a split-transaction bus.

The key addition is an eight-entry *request table* that keeps track of all outstanding requests on the bus. Whenever a new request is issued on the bus, it is added at the same index in all processors' request table using a common assignment algorithm. The request table serves many purposes, including request merging, and ensuring that only a single request can be outstanding for any given memory block.

only an address but also a cache block of data and is therefore large. The controller limits the number of requests it has outstanding so that there is buffer space available for every response.

Flow control is also needed at main memory. Each of the (eight) pending requests can generate a writeback that main memory must accept. Since writeback transactions do not require a response, they can happen in quick succession on the bus, possibly overflowing buffers in the main memory subsystem. For flow control, the Challenge design provides separate NACK (negative acknowledgment) lines for the address and data portions of bus, since the bus allows independent arbitration for each portion. Before a request or response sub-transaction has reached its ack phase and completed, any other processor or the main memory can assert a NACK signal, for example if it finds its buffers full. The sub-transaction is then canceled everywhere, and must be retried. One common option, used in the Challenge, is to have the requestor retry periodically until it succeeds. Backoff and priorities can be used to reduce bandwidth consumption for failed retries and to avoid starvation. The Sun Enterprise uses an interesting alternative for writes that encounter a full buffer. In this case, the destination buffer—which could not accommodate the

data from the write on the first attempt—itself initiates the retry when it has enough buffer space. The original writer simply keeps watch for the retry transaction on the bus, and places the data on the data bus. The operation of the Enterprise bus ensures that the space in the destination buffer is still available when the data arrive. This guarantees that writes will succeed with only one retry bus transaction, and does not require priority-based arbitration which can slow the bus down.

6.5.5 Path of a Cache Miss

Given this design, we are ready to examine how various requests may be handled, and the race conditions that might occur. Let us first look at the case where a processor has a read miss in the cache, so the request part of a BusRd transaction should be generated. The request first checks the currently pending entries in the request table. If it finds one with a matching address, there are two possible courses of action depending on the nature of the pending request:

1. The earlier request was a BusRd request for the same block. This is great news for this processor, since the request needn't be put on the bus but can just grab the data when the response to the earlier request appears on the bus. To accomplish this, we add two new bits to each entry in the request-table which say: (i) do I wish to grab the data response for this request, and (ii) am I the original generator of this request. In our situation these bits will be set to 1 and 0 respectively. The purpose of the first bit is obvious; the purpose of the second bit is to help determine in which state (valid-exclusive versus shared) to signal the data response. If a processor is not the original requestor, then it must assert the sharing line on the snoop bus when it grabs the response data from the bus, so that all caches will load this block in shared state and not valid-exclusive. If a processor *is* the original requestor, it does not assert the sharing line when it grabs the response from the bus, and if the sharing line is not asserted at all then it will grab the block in valid-exclusive state.

2. The earlier request was incompatible with a BusRd, for example, a BusRdX. In this case, the controller must hold on to the request until it sees a response to the previous request on the bus, and only then attempt the request. The "processor-side" controller is typically responsible for this.

If the controller finds that there are no matching entries in the request-table, it can go ahead and issue the request on the bus. However, it must watch out for race conditions. For example, when the controller first examines the request table it may find that there are no conflicting requests, and so it may request arbitration for the bus. However, before it is granted the bus, a conflicting request may appear on the bus, and then it may be granted the very next use of the bus. Since this design does not allow conflicting requests on the bus, when the controller sees a conflicting request in the slot just before its own it should (i) issue a null request (a no-action request) on the bus to occupy the slot it had been granted and (ii) withdraw from further arbitration until a response to the conflicting request has been generated.

Suppose the processor does manage to issue the BusRd request on the bus. What should other cache controllers and the main memory controller do? The request is entered into the request tables of all cache controllers, including the one that issued this request, as soon as it appears on the bus. The controllers start checking their caches for the requested memory block. The main memory subsystem has no idea whether this block is dirty in one of the processor's caches, so it independently starts fetching this block. There are now three different scenarios to consider:

1. One of the caches may determine that it has the block dirty, and may acquire the bus to generate a response before main memory can respond. On seeing the response on the bus, main

memory simply aborts the fetch that it had initiated, and the cache controllers that are waiting for this block grab the data in a state based on the value of the snooping lines. If a cache controller has not finished snooping by the time the response appears on the bus, it will keep the inhibit line asserted and the response transaction will be extended (i.e. will stay on the bus). Main memory also receives the response since the block was dirty in a cache. If main memory does not have the buffer space needed, it asserts the NACK signal provided for flow control, and it is the responsibility of the controller holding the block dirty to retry the response transaction later.

2. Main memory may fetch the data and acquire the bus before the cache controller holding the block dirty has finished its snoop and/or acquired the bus. The controller holding the block dirty will first assert the inhibit line until it has finished its snoop, and then assert the dirty line and release the inhibit line, indicating to the memory that it has the latest copy and that memory should not actually put its data on the bus. On observing the dirty line, memory cancels its response transaction and does not actually put the data on the bus. The cache with the dirty block will sometime later acquire the bus and put the data response on it.

3. The simplest scenario is that no other cache has the block dirty. Main memory will acquire the bus and generate the response. Cache controllers that have not finished their snoop will assert the inhibit line when they see the response from memory, but once they de-assert it memory can supply the data (Cache-to-cache sharing is not used for data in shared state).

Processor writes are handled similarly to reads. If the writing processor does not find the data in its cache in a valid state, a BusRdX is generated. As before, it checks the request table and then goes on the bus. Everything is the same as for a bus read, except that main memory will not sink the data response if it comes from another cache (since it is going to be dirty again) and no other processor can grab the data. If the block being written is valid but in shared state, a BusUpgr is issued. This requires no response transaction (the data is known to be in main memory as well as in the writer's cache); however, if any other processor was just about to issue a BusUpgr for the same block, it will need to now convert its request to a BusRdX as in the atomic bus.

6.5.6 Serialization and Sequential Consistency

Consider serialization to a single location. If an operation appearing on the bus is a read (miss), no subsequent write appearing on the bus after the read should be able to change the value returned to the read. Despite multiple outstanding transactions on the bus, here this is easy since conflicting requests to the same location are not allowed simultaneously on the bus. If the operation is a BusRdX or BusUpgr generated by a write operation, the requesting cache will perform the write into the cache array after the response phase; subsequent (conflicting) reads to the block are allowed on the bus only after the response phase, so they will obtain the new value. (Recall that the response phase may be a separate action on the bus as in a BusRdX or may be implicitly generated once the request wins arbitration as in a BusUpgr).

Now consider the serialization of operations to different locations needed for sequential consistency. The logical total order on bus transactions is established by the order in which requests are granted for the address bus. Once a BusRdX or BusUpgr has obtained the bus, the associated write is committed. However, with multiple outstanding requests on the bus, the invalidations are buffered and it may be a while before they are applied to the cache. Commitment of a write does not guarantee that the value produced by the write is already visible to other processors; only actual completion guarantees that. The separation between commitment and completion and the need for buffering multiple outstanding transactions imply the need for further mechanisms to

ensure that the necessary orders are preserved between the bus and the processor. The following examples will help make this concrete.

Example 6-3   Consider the two code fragments shown below. What results for (A,B) are disallowed under SC? Assuming a single level of cache per processor and multiple outstanding transactions on the bus, and no special mechanisms to preserve orders between bus and cache or processor, show how the disallowed results may be obtained. Assume an invalidation-based protocol, and that the initial values of A and B are 0.

P1	P2
A = 1	rd B
B = 1	rd A

P1	P2
A = 1	B = 1
rd B	rd A

Answer   In the first example, on the left, the result not permitted under SC is (A,B) = (0,1). However, consider the following scenario. P1's write of A commits, so it continues with the write of B (under the sufficient conditions for SC). The invalidation corresponding to B is applied to the cache of P2 before that corresponding to A, since they get reordered in the buffers. P2 incurs a read miss on B and obtains the new value of 1. However, the invalidation for A is still in the buffer and not applied to P2's cache even by the time P2 issues the read of A. The read of A is a hit, and completes returning the old value 0 for A from the cache.

In the example on the right, the disallowed result is (0,0). However, consider the following scenario. P1 issues and commits its write of A, and then goes forward and completes the read of B, reading in the old value of 0. P2 then writes B, which commits, so P2 proceeds to read A. The write of B appears on the bus after the write of A, so they should be serialized in that order and P2 should read the new value of A. However, the invalidation corresponding to the write of A by P1 is sitting in P2's incoming buffer and has not yet been applied to P2's cache. P2 sees a read hit on A and completes returning the old value of A which is 0.

With commitment substituting for commitment and multiple outstanding operations being buffered between bus and processor, the key property that must be preserved for sequential consistency is the following: A processor should not be allowed to actually see the new value due to a write before previous writes (in bus order, as usual) are visible to it. There are two ways to preserve this property: by not letting certain types of incoming transactions from bus to cache be reordered in the incoming queues, and by allowing these reorderings but ensuring that the necessary orders are preserved when necessary. Let us examine each approach briefly.

A simple way to follow the first approach is to ensure that all incoming transactions from the bus (invalidations, read miss replies, write commitment acknowledgments etc.) propagate to the processor in FIFO order. However, such strict ordering is not necessary. Consider an invalidation-based protocol. Here, there are two ways for a new value to be brought into the cache and made available to the processor to read without another bus operation. One is through a read miss, and the other is through a write by the same processor. On the other hand, writes from other proces-

sors become *visible* to a processor (even though the values are not yet local) when the corresponding invalidations are applied to the cache. The invalidations due to writes that are previous to the read miss or local write that provides the new value are already in the queue when the read miss or local write appears on the bus, and therefore are either in the queue or applied to the cache when the reply comes back. All we need to ensure, therefore, is that a reply (read miss or write commitment) does not overtake an invalidation between the bus and the cache; i.e. all previous invalidations are applied before the reply is received.

Note that incoming invalidations may be reordered with regard to one another. This is because the new value corresponding to an invalidation is seen only through the corresponding read miss, and the read miss reply is not allowed to be reordered with respect to the previous invalidation. In an update-based protocol, the new value due to a write does not require a read miss and reply, but rather can be seen as soon as the incoming update has been applied. Writes are made visible through updates as well. This means that not only should replies not overtake updates, but updates should not overtake updates either.

An alternative is to allow incoming transactions from the bus to be reordered on their way to the cache, but to simply ensure that all previously committed writes are applied to the cache (by flushing them from the incoming queue) before an operation from the local processor that enables it to see a new value can be completed. After all, what really matters is not the order in which invalidations/updates are applied, but the order in which the corresponding new values are seen by the processor. There are two natural ways to accomplish this. One is to flush the incoming invalidations/updates every time the processor tries to complete an operation that may enable it to see a new value. In an invalidation-based protocol, this means flushing before the processor is allowed to complete a read miss or a write that generates a bus transaction; in an update-based protocol, it means flushing on every read operation as well. The other way is to flush under the following circumstance: Flush whenever a processor is about to access a value (complete a read hit or miss) if it has indeed seen a new value since the last flush, i.e. if a reply or an update has been applied to the cache since the last flush. The fact that operations are reordered from bus to cache and a new value has been applied to the cache means that there may be invalidations or updates in the queue that correspond to writes that are previous to that new value; those writes should now be applied before the read can complete. Showing that these techniques disallow the undesirable results in the example above is left as an exercise that may help make the techniques concrete. As we will see soon, the extension of the techniques to multi-level cache hierarchies is quite natural.

Regardless of which approach is used, write atomicity is provided naturally by the broadcast nature of the bus. Writes are committed in the same order with respect to all processors, and a read cannot see the value produced by a write until that write has committed with respect to all processors. With the above techniques, we can substitute complete for commit in this statement, ensuring atomicity. We will discuss deadlock, livelock and starvation issues introduced by a split transaction bus after we have also introduced multi-level cache hierarchies in this context. First, let us look at some alternative approaches to organizing a protocol with a split transaction bus.

6.5.7 Alternative design choices

There are reasonable alternative positions for all of request-response ordering, dealing with conflicting requests, and flow control. For example, ensuring that responses are generated in order with respect to requests—as cache controllers are inclined to do—would simplify the design. The

fully-associative request table could be replaced with a simple FIFO buffer that stores pending requests that were observed on the bus. As before, a request is put into the FIFO only when the request actually appears on the bus, ensuring that all entities (processors and memory system) have exactly the same view of pending requests. The processors and the memory system process requests in FIFO order. At the time the response is presented, as in the earlier design, if others have not completed their snoops they assert the inhibit line and extend the transaction duration. That is, snoops are still reported together with responses. The difference is in the case where the memory generates a response first even though a processor has that block dirty in its cache. In the previous design, the cache controller that had the block dirty released the inhibit line and asserted the dirty line, and arbitrated for the bus again later when it had retrieved the data. But now to pre-serve the FIFO order this response has to be placed on the bus before any other request is ser-viced. So the dirty controller does not release the inhibit line, but extends the current bus transaction until it has fetched the block from its cache and supplied it on the bus. This does not depend on anyone else accessing the bus, so there is no deadlock problem.

While FIFO request-response ordering is simpler, it can have performance problems. Consider a multiprocessor with an interleaved memory system. Suppose three requests A, B, and C are issued on the bus in that order, and that A and B go to the same memory bank while C goes to a different memory bank. Forcing the system to generate responses in order means that C will have to wait for both A and B to be processed, though data for C will be available much before data for B is available because of B's bank conflict with A. This is the major motivation for allowing out of order responses, since caches are likely to respond to requests in order anyway.

Keeping responses in order also makes it more tractable to allow conflicting requests to the same block to be outstanding on the bus. Suppose two BusRdX requests are issued on a block in rapid succession. The controller issuing the latter request will invalidate its block, as before. The tricky part with a split-transaction bus is that the controller issuing the earlier request sees the latter request appear on the bus before the data response that it awaits. It cannot simply invalidate its block in reaction to the latter request, since the block is in flight and its own write needs to be performed before a flush or invalidate. With out-of-order responses, allowing this conflicting request may be difficult. With in-order responses, the earlier requestor knows its response will appear on the bus first, so this is actually an opportunity for a performance enhancing optimiza-tion. The earlier-requesting controller responds to the latter request as usual, but notes that the latter request is pending. When its response block arrives, it updates the word to be written and "short-cuts" the block back out to the bus, leaving its own block invalid. This optimization reduces the latency of ping-ponging a block under write-write false sharing.

If there is a fixed delay from request to snoop result, conflicting requests can be allowed even without requiring data responses to be in order. However, since conflicting requests to a block go into the same queue at the desitation they themselves are usually responded to in order anyway, so they can be handled using the shortcut method described above (this is done in the Sun Enter-prise systems). It is left as an exercise to think about how one might allow conflicting requests with fixed-delay snoop results when responses to them may appear on the bus out of order.

In fact, as long as there is a well-defined order among the request transactions, they do not even need to be issued sequentially on the same bus. For example, the Sun SPARCcenter 2000 used two distinct split-phase busses and the Cray 6400 used four to improve bandwidth for large con-figurations. Multiple requests may be issued on a single cycle. However, a priority is established among the busses so that a logical order is defined even among the concurrent requests.

6.5.8 Putting it together with multi-level caches

We are now ready to combine the two major enhancements to the basic protocol from which we started: multi-level caches and a split-transaction bus. The base design is a (Challenge-like) split-transaction bus and a two-level cache hierarchy. The issues and solutions generalize to deeper hierarchies. We have already seen the basic issues of request, response, and invalidation propagation up and down the hierarchy. The key new issue we need to grapple with is that it takes a considerable number of cycles for a request to propagate through the cache controllers. During this time, we must allow other transactions to propagate up and down the hierarchy as well. To maintain high bandwidth while allowing the individual units to operate at their own rates, queues are placed between the units. However, this raises a family of questions related to deadlock and serialization.

A simple multi-level cache organization is shown in Figure 6-10. Assume that a processor can have only one request outstanding at a time, so there are no queues between the processor and first-level cache. A read request that misses in the L1 cache is passed on to the L2 cache (1). If it misses there, a request is placed on the bus (2). The read request is captured by all other cache controllers in the incoming queue (3). Assuming the block is currently in modified state in the L1 cache of another processor, the request is queued for L1 service (4). The L1 demotes the block to shared and flushes it to the L2 cache (5), which places it on the bus (6). The response is captured by the requestor (7) and passed to the L1 (8), whereupon the word is provided to the processor.
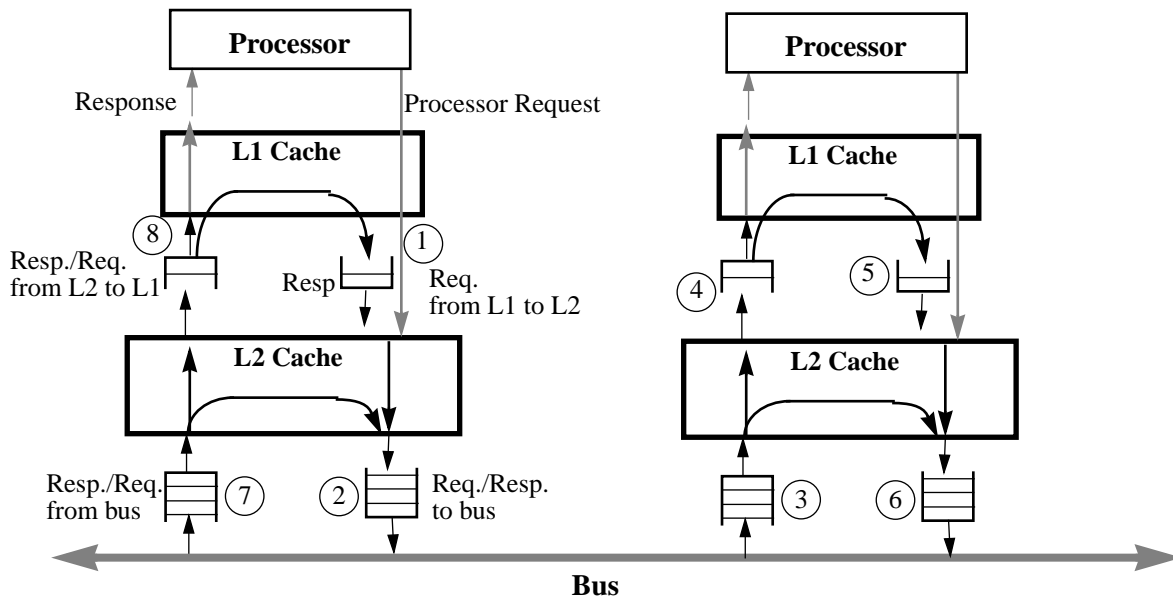
Figure 6-10 Internal queues that may exist inside a multi-level cache hierarchy.

One of the concerns with such queue structures is deadlock. To avoid the fetch deadlock problem discussed earlier, as before an L2 cache needs to be able to buffer incoming requests or responses while it has a request outstanding, so that the bus may be freed up. With one outstanding request

per processor, the incoming queues between the bus and the L2 cache need to be larger enough to hold a request from each other processor (plus a response to its request). That takes care of the case where all processors make a request of this cache while the processor has a request outstanding. If they are made smaller than this to conserve real-estate (or if there are multiple outstanding requests per processor), it is necessary to NACK bus requests when there is not room to enqueue them. Also, one slot in the bus to L2 and in the L2 to L1 queues is reserved for the response to the processor's outstanding request, so that each processor can always drain its outstanding responses. If NACKs are used, the request bus arbitration needs to include a mechanism to ensure forward progress under heavy contention, such as a simple priority scheme.

In addition to fetch deadlock, buffer deadlock can occur within the multilevel cache hierarchy as well. For example, suppose there is a queue in each direction between the L1 and L2 cache, both of which are writeback caches, and each queue can hold two entries. It is possible that the L1->L2 queue holds two outgoing read requests, which can be satisfied in the L2 cache but will generate replies to L1, and the L2->L1 queue holds two incoming read requests, which can be satisfied in the L1 cache. We now have a classical circular buffer dependence, and hence deadlock. Note that this problem occurs only in hierarchies in which there is more than one level of writeback cache, i.e. in which a cache higher than the one closest to the bus is writeback. Otherwise, incoming requests do not generate replies from higher level caches, so there is no circularity and no deadlock problem (recall that invalidations are acknowledged implicitly from the bus itself, and do not need acknowledgments from the caches).

One way to deal with this buffer deadlock problem in a multilevel writeback cache hierarchy is to limit the number of outstanding requests from processors and then provide enough buffering for incoming requests and responses at each level. However, this requires a lot of real estate and is not scalable (each request may need two outgoing buffer entries, one for the request and one for the writeback it might generate, and with a large number of outstanding bus transactions the incoming buffers may need to have many entries as well) . An alternative way uses a general deadlock avoidance technique for situations with limited buffering, which we will discuss in the context of systems with physically distributed memory in the next chapter, where the problem is more acute. The basic idea is to separate transactions (that flow through the hierarchy) into requests and responses. A transaction can be classified as a response if it does not generate any further transactions; a request may generate a response, but not transaction may generate another request (although a request may be transferred to the next level of the hierarchy if it does not generate a reply at the original level). With this classification, we can avoid deadlock if we provide separate queues for requests and responses in each direction, and ensure that responses are always extracted from the buffers. After we have discussed this technique in the next chapter, we shall apply to this particular situation with multilevel writeback caches in the exercises.

There are other potential deadlock considerations that we may have to consider. For example, with a limited number of outstanding transactions on the bus, it may be important for a response from a processor's cache to get to the bus before new outgoing requests from the processor are allowed. Otherwise the existing requests may never be satisfied, and there will be no progress. The outgoing queue or queues must be able to support responses bypassing requests when necessary.

The second major concern is maintaining sequential consistency. With multi-level caches, it is all the more important that the bus not wait for an invalidation to reach all the way up to the first-level cache and return a reply, but consider the write committed when it has been placed on the bus and hence in the input queue to the lowest-level cache. The separation of commitment and

completion is even greater in this case. However, the techniques discussed for single level caches extend very naturally to this case: We simply apply them at each level of the cache hierarchy. Thus, in an invalidation based protocol the first solution extends to ensuring at each level of the hierarchy that replies are not reordered with respect to invalidations in the incoming queues to that level (replies from a lower level cache to a higher-level cache are treated as replies too for this purpose). The second solution extends to either not letting an outgoing memory operation proceed past a level of the hierarchy before the incoming invalidations to that level are applied to that cache, or flushing the incoming invalidations at a level if a reply has been applied to that level since the last flush.

6.5.9 Supporting Multiple Outstanding Misses from a Processor

Although we have examined split transaction buses, which have multiple transactions outstanding on them at a time, so far we have assumed that a given processor can have only one memory request outstanding at a time. This assumption is simplistic for modern processors, which permit multiple outstanding requests to tolerate the latency of cache misses even on uniprocessor systems. While allowing multiple outstanding references from a processor improves performance, it can complicate semantics since memory accesses from the same processor may complete in a different order in the memory system than that in which they were issued.

One example of multiple outstanding references is the use of a write buffer. Since we would like to let the processor proceed to other computation and even memory operations after it issues a write but before it obtains exclusive ownership of the block, we put the write in the write buffer. Until the write is serialized, it should be visible only to the issuing processor and not to other processors, since otherwise it may violate write serialization and coherence. One possibility is to write it into the local cache but not make it available to other processors until exclusive ownership is obtained (i.e. not let the cache respond to requests for it until then). The more common approach is to keep it in the write buffer and put it in the cache (making it available to other processors through the bus) only when exclusive ownership is obtained.

Most processors use write buffers more aggressively, issuing a sequence of writes in rapid succession into the write buffer without stalling the processor. In a uniprocessor this approach is very effective, as long as reads check the write buffer. The problem in the multiprocessor case is that, in general, the processor cannot be allowed to proceed with (or at least complete) memory operations past the write until the exclusive ownership transaction for the block has been placed on the bus. However, there are special cases where the processor can issue a sequence of writes without stalling. One example is if it can be determined that the writes are to blocks that are in the local cache in modified state. Then, they can be buffered between the processor and the cache, as long as the cache processes the writes before servicing a read or exclusive request from the bus side. There is also an important special case in which a sequence of writes can be buffered regardless of the cache state. That is where the writes are all to the same block and no other memory operations are interspersed between those writes. The writes may be coalesced while the controller is obtaining the bus for the read exclusive transaction. When that transaction occurs, it makes the entire sequence of writes visible at once. The behavior is the same as if the writes were performed after the bus transaction, but before the next one. Note that there is no problem with sequences of writebacks, since the protocol does not require them to be ordered.

More generally, to satisfy the sufficient conditions for sequential consistency, a processor having the ability to proceed past outstanding write and even read operations raises the question of who

should wait to "issue" an operation until the previous one in program order completes. Forcing the processor itself to wait can eliminate any benefits of the sophisticated processor mechanisms (such as write buffers and out-of-order execution). Instead, the buffers that hold the outstanding operations— such as the write buffer and the reorder buffer in dynamically scheduled out-of-order execution processors—can serve this purpose. The processor can issue the next operation right after the previous one, and the buffers take charge of not making write operations visible to the memory and interconnect systems (i.e. not issuing them to the externally visible memory system) or not allowing read operations to complete out of program order with respect to outstanding writes even though the processor may issue and execute them out of order. The mechanisms needed in the buffers are often already provided to provide precise interrupts, as we will see in later chapters. Of course, simpler processors that do not proceed past reads or writes make it easier to maintain sequential consistency. Further semantic implications of multiple outstanding references for memory consistency models will be discussed in Chapter 9, when we examine consistency models in detail.

From a design perspective, exploiting multiple outstanding references most effectively requires that the processor caches allow multiple cache misses to be outstanding at a time, so that the latencies of these misses can be overlapped. This in turn requires that either the cache or some auxiliary data structure keep track of the outstanding misses, which can be quite complex since they may return out of order. Caches that allow multiple outstanding misses are called *lockup-free* caches [Kro81,Lau94], as opposed to *blocking* caches that allow only one outstanding miss. We shall discuss the design of lockup-free caches when we discuss latency tolerance in Chapter 11.

Finally, consider the interactions with split transaction busses and multi-level cache hierarchies. Given a design that supports a split-transaction bus with multiple outstanding transactions and a multi-level cache hierarchy, the extensions needed to support multiple outstanding operations per processor are few and are mostly for performance. We simply need to provide deeper request queues from the processor to the bus (the request queues pointing downwards in Figure 6-10), so that the multiple outstanding requests can be buffered and not stall the processor or cache. It may also be useful to have deeper response queues, and more write-back and other types of buffers, since there is now more concurrency in the system. As long as deadlock is handled by separating requests from replies, the exact length of any of these queues is not critical for correctness. The reason for such few changes is that the lockup-free caches themselves perform the complex task of merging requests and managing replies, so to the caches and the bus subsystem below it simply appears that there are multiple requests to distinct blocks coming from the processor. While some potential deadlock scenarios might become exposed that would not have arisen with only one outstanding request per processor—for example, we may now see the situation where the number of requests outstanding from all processors is more than the bus can take, so we have to ensure responses can bypass requests on the way out—the support discussed earlier for split-transaction buses makes the rest of the system capable of handling multiple requests from a processor without deadlock.

6.6 Case Studies: SGI Challenge and Sun Enterprise SMPs

This section places the general design and implementation issues discussed above into a concrete setting by describing two multiprocessor systems, the SGI Challenge and the Sun Enterprise 6000.

The SGI Challenge is designed to support up to 36 MIPS R4400 processors (peak 2.7 GFLOPS) or up to 18 MIPS R8000 processors (peak 5.4 GFLOPS) in the Power Challenge model. Both systems use the same system bus, the Powerpath-2 bus, which provides a peak bandwidth of 1.2 Gbytes/sec and the system supports up to 16 Gbytes of 8-way interleaved main memory. Finally, the system supports up to 4 PowerChannel-2 I/O buses, each providing a peak bandwidth of 320 Mbytes/sec. Each I/O bus in turn can support multiple Ethernet connections, VME/SCSI buses, graphics cards, and other peripherals. The total disk storage on the system can be several Terabytes. The operating system that runs on the hardware is a variant of SVR4 UNIX called IRIX; it is a symmetric multiprocessor kernel in that any of the operating systems' tasks can be done on any of the processors in the system. Figure 6-11 shows a high-level diagram of the SGI Challenge system.
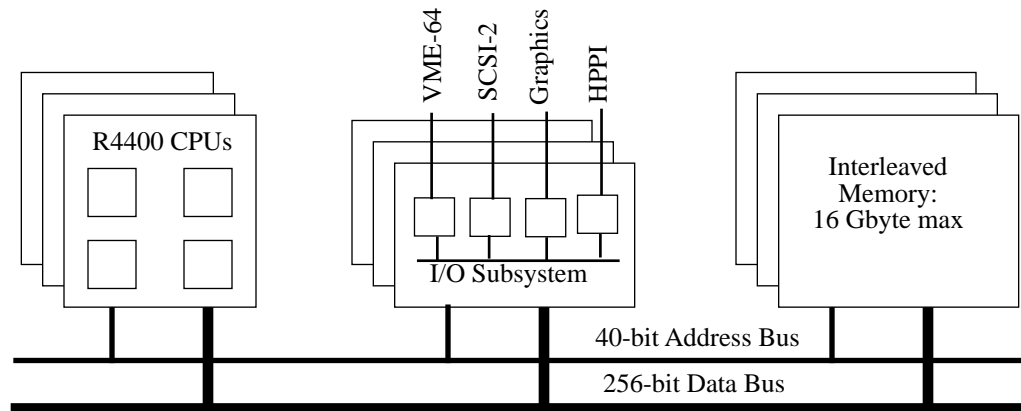
Figure 6-11 The SGI Challenge multiprocessor

With four processors per board, the thirty-six processors consume nine bus slots. It can support up to 16 Gbytes of 8-way interleaved main memory. The I/O boards provide a separate 320 Mbytes/sec I/O bus, to which other standard buses and devices interface. The system bus has a separate 40-bit address path and a 256-bit data path, and supports a peak bandwidth of 1.2 Gbytes/sec. The bus is split-transaction and up to eight requests can be outstanding on the bus at any given time.

The Sun Enterprise 6000 is designed to support up to 30 UltraSPARC processors (peak 9 GFLOPs). The Gigaplane^{TM} system bus provides a peak bandwidth of 2.67 GB/s (83.5MHz times 32 bytes), and the system can support up to 30 GB of up to 16-way interleaved memory. The 16 slots in the machine can be populated with a mix of processing boards and I/O boards, as long as there is a least one of each. Each processing board has two CPU modules and two (512 bit wide) memory banks of up to 1 GB each, so the memory capacity and bandwidth scales with the number of processors. Each I/O card provides two independent 64-bit x 25 MHz SBUS I/O busses, so the I/O bandwidth scales with the number of I/O cards up to more than 2 GB/s. The

total disk storage can be tens of terabytes. The operating system is Solaris UNIX. Figure 6-12 shows a block diagram of the Sun Enterprise system.
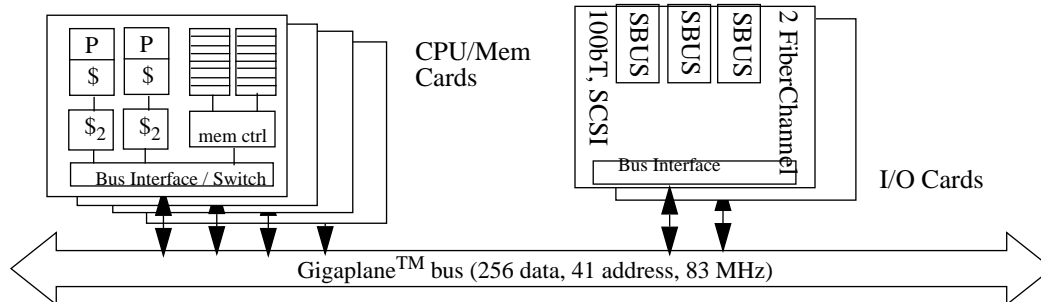
Figure 6-12 The Sun Enterprise 6000 multiprocessor

6.6.1 SGI Powerpath-2 System Bus

The system bus forms the core interconnect for all components in the system. As a result, its design is affected by requirements of all other components, and design choices made for it, in turn, affect back the design of other components. The design choices for buses include: multiplexed versus non-multiplexed address and data buses, wide (e.g., 256 or 128-bit) versus narrower (64-bit) data bus, the clock rate of the bus (affected by signalling technology used, length of bus, the number of slots on bus), split-transaction versus non-split-transaction design, the flow-control strategy, and so on. The powerpath-2 bus is non-multiplexed with 256-bit wide data portion and 40-bit wide address portion, it is clocked at 47.6 MHz, and it is split-transaction supporting 8 outstanding read requests. While the very wide data path implies that the cost of connecting to the bus is higher (it requires multiple bit-sliced chips to interface to it), the benefit is that the high bandwidth of 1.2 Gbytes/sec can be achieved at a reasonable clock rate. At the 50MHz clock rate the bus supports sixteen slots, nine of which can be populated with 4-processor boards to obtain a 36-processor configuration. The width of the bus also affects (and is affected by) the cache block size chosen for the machine. The cache block size in the Challenge machine is 128 bytes, implying that the whole cache block can be transferred in only four bus clocks; a much smaller block size would have resulted in less effective use of the bus pipeline or a more complex design. The bus width affects many other design decisions. For example, the individual board is fairly large in order to support such a large connector. The bus interface occupies roughly 20% of the board, in a strip along the edge, making it natural to place four processors on each board.

Let us look at the Powerpath-2 bus design in a little more detail. The bus consists of a total of 329 signals: 256 data, 8 data parity, 40 address, 8 command, 2 address+command parity, 8 data-resource ID, and 7 miscellaneous. The types and variations of transactions on the bus is small, and all transactions take exactly 5 cycles. System wide, bus controller ASICs execute the following 5-state machine synchronously: arbitration, resolution, address, decode, and acknowledge. When no transactions are occurring, each bus controller drops into a 2-state idle machine. The 2-state idle machine allows new requests to arbitrate immediately, rather than waiting for the arbitrate state to occur in the 5-state machine. Note that 2-states are required to prevent different

requestors from driving arbitration lines on successive cycles. Figure 6-13 shows the state machine underlying the basic bus protocol.
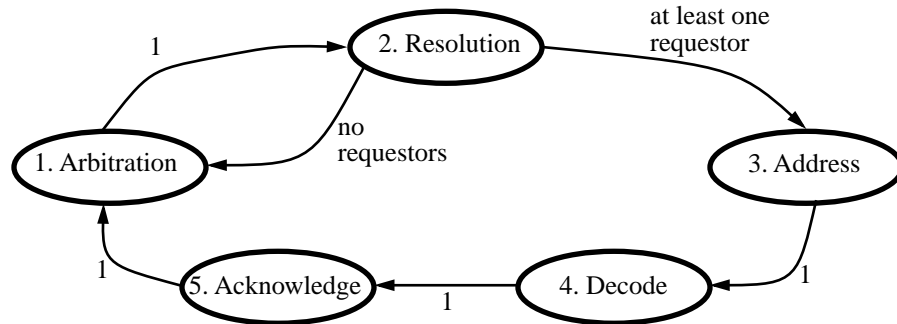
Figure 6-13 Powerpath-2 bus state-transition diagram.

The bus interfaces of all boards attached to the system-bus synchronously cycle through the five states shown in the figure; this is also the duration of all address and data transactions on the bus. When the bus is idle, however, it only loops between states 1 and 2.

Since the bus is split transaction, the address and data buses must be arbitrated for separately. In the arbitration cycle, the 48 address+command lines are used for arbitration. The lower 16 lines are used for data bus arbitration and the middle 16 lines are used for address bus arbitration. For transactions that require both address and data buses together, e.g., writebacks, corresponding bits for both buses can be set high. The top 16 lines are used to make *urgent* or high-priority requests. Urgent requests are used to avoid starvation, for example, if a processor times-out waiting to get access to the bus. The availability of urgent-type requests allowed the designers considerable flexibility in favoring service of some requests over others for performance reasons (e.g., reads are given preference over writes), while still being confident that no requestor will get starved.

At the end of the arbitration cycle, all bus-interface ASICs capture the 48-bit state of the requestors. A distributed arbitration scheme is used, so every bus master sees all of the bus requests, and in the resolution cycle, each one independently computes the same winner. While distributed arbitration consumes more of ASIC's gate resources, it saves the latency incurred by a centralized arbitrator of communicating winners to everybody via bus grant lines.

During the address cycle, the address-bus master drives the address and command buses with corresponding information. Simultaneously, the data-bus master drives the *data resource ID* line corresponding to the response. (The data resource ID corresponds to the global tag assigned to this read request when it was originally issued on the bus. Also see Section 6.5 for details.)

During the decode cycle, no signals are driven on the address bus. Internally, each bus-interface slot decides how to respond to this transaction. For example, if the transaction is a writeback, and the memory system currently has insufficient buffer resources to accept the data, in this cycle it will decide that it must NACK (negative acknowledge or reject) this transaction on the next cycle, so that the transaction can be retried at a later time. In addition all slots prepare to supply the proper cache-coherence information.

During the data acknowledge cycle, each bus interface slot responds to the recent data/address bus transaction. The 48 address+command lines are used as follows. The top 16 lines indicate if the device in the corresponding slot is rejecting the address-bus transaction due to insufficient buffer space. Similarly, the middle 16 lines are used to possibly reject the data-bus transaction. The lowest 16 lines indicate the cache-state of the block (present vs. not-present) being transferred on the data-bus. These lines help determine the state in which the data block will be loaded in the requesting processor, e.g., valid-exclusive versus shared. Finally, in case one of the processors has not finished its snoop by this cycle, it indicates so by asserting the corresponding inhibit line. (The data-resource ID lines during the data acknowledgment and arbitration cycles are called inhibit lines.) It continues to assert this line until it has finished the snoop. If the snoop indicates a clean cache block, the snooping node simply drops the inhibit line, and allows the requesting node to accept memory's response. If the snoop indicated a dirty block, the node re-arbitrates for the data bus and supplies the latest copy of the data, and only then drops the inhibit line.

For data-bus transactions, once a slot becomes the master, the 128 bytes of cache-block data is transferred in four consecutive cycles over the 256-bit wide data path. This four-cycle sequence begins with the data acknowledgment cycle and ends at the address cycle of the following transaction. Since the 256-bit wide data path is used only for four out of five cycles, the maximum possible efficiency of these data lines is 80%. In some sense though, this is the best that could be done; the signalling technology used in the Powerpath-2 bus requires one cycle turn-around time between different masters driving the lines. Figure 6-13 shows the cycles during which various bus lines are driven and their semantics.
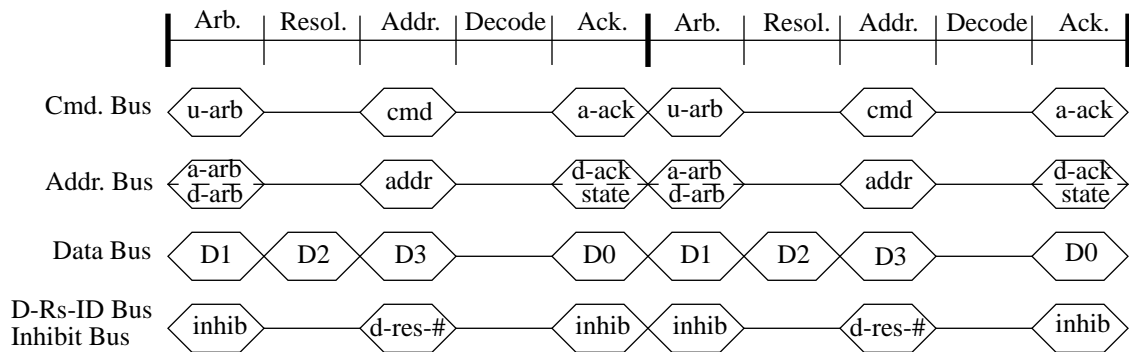
Figure 6-14 Powerpath-2 bus timing diagram.

During the arbitration cycle, the 48-bits of the address + command bus indicate requests from the 16-bus slots for data transaction, address transaction, and urgent transactions. Each bus interface determines the results of the arbitration independently following a common algorithm. If an address request is granted, the address +command are transferred in the address cycle, and the requests can be NACK'ed in the acknowedgment cycle. Similarly, if a data request is granted, the tag associated with it (data-resource-id) is transferred in the address cycle, it can be NACK'ed in the ack cycle, and the data is transferred in the following D0-D3 cycles.

6.6.2 SGI Processor and Memory Subsystems

In the Challenge system, each board can contain 4 MIPS R4400 processors. Furthermore, to reduce the cost of interfacing to the bus, many of the bus-interface chips are shared between the processors. Figure 6-13 shows the high-level organization of the processor board.
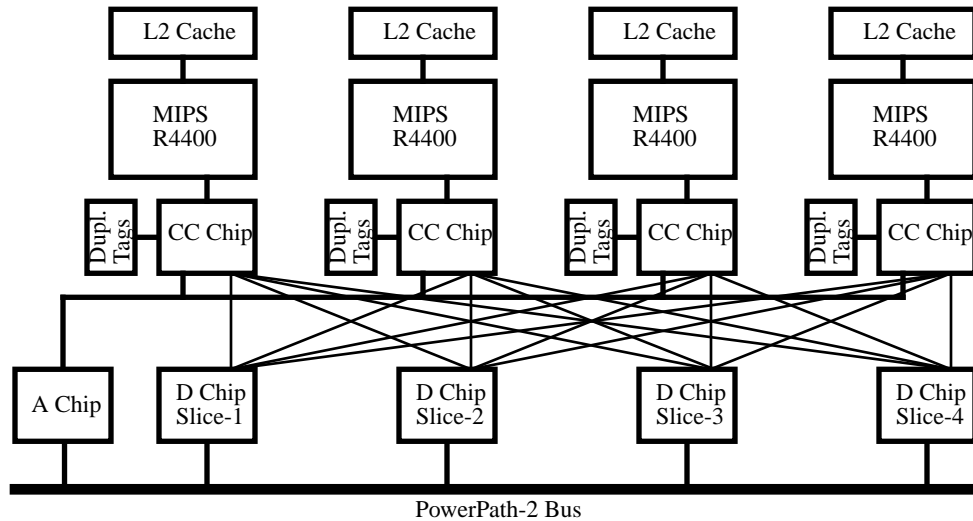
Figure 6-15 Organization and chip partitioning of the SGI Challenge processor board.

To minimize number of bus slots required to support thirty-six processors, four processors are put on each board. To maintain coherence and to interface to the bus, there is one cache-coherence chip per processor, there is one shared A-chip that keeps track of requests from all four processors, and to interface to the 256-bit wide data bus, there are four shared bit-sliced D-chips.

The processor board uses three different types of chips to interface to the bus and to support cache coherence. There is a single A-chip for all four processors that interfaces to the address bus. It contains logic for distributed arbitration, the eight-entry request-table storing currently outstanding transactions on the bus (see Section 6.5 for details), and other control logic for deciding when transactions can be issued on the bus and how to respond to them. It passes on requests observed on the bus to the CC-chip (one for each processor), which uses a duplicate set of tags to determine the presence of that memory block in the local cache, and communicates the results back to the A-chip. All requests from the processor also flow through the CC-chip to the A-chip, which then presents them on the bus. To interface to the 256-bit wide data bus, four bit-sliced D-chips are used. The D-chips are quite simple and are shared among the processors; they provide limited buffering capability and simply pass data between the bus and the CC-chip associated with each processor.

The Challenge main memory subsystem uses high-speed buffers to fan out addresses to a 576-bit wide DRAM bus. The 576 bits consist of 512 bits of data and 64 bits of ECC, allowing for single bit in-line correction and double bit error detection. Fast page-mode access allows an entire 128 byte cache block to be read in two memory cycles, while data buffers pipeline the response to the 256-bit wide data bus. Twelve clock cycles (~250ns) after the address appears on the bus, the response data appears on the data bus. Given current technology, a single memory board can hold 2 Gbytes of memory and supports a 2-way interleaved memory system that can saturate the 1.2 Gbytes/sec system bus.

Given the raw latency of 250ns that the main-memory subsystem takes, it is instructive to see the overall latency experienced by the processor. On the Challenge this number is close to 1µs or 1000ns. It takes approximately 300ns for the request to first appear on the bus; this includes time taken for the processor to realize that it has a first-level cache miss, a second-level cache miss, and then to filter through the CC-chip down to the A-chip. It takes approximately another 400ns for the complete cache block to be delivered to the D-chips across the bus. These include the 3 bus cycles until the address stage of the request transaction, 12 cycles to access the main memory, and another 5 cycles for the data transaction to deliver the data over the bus. Finally, it takes another 300ns for the data to flow through the D-chips, through the CC-chip, through the 64-bit wide interface onto the processor chip (16 cycles for the 128 byte cache block), loading the data into the primary cache and restart of the processor pipeline.[1]

To maintain cache coherence, by default the SGI Challenge used the Illinois MESI protocol as discussed earlier. It also supports update transactions. Interactions of the cache coherence protocol and the split-transaction bus interact are as described in Section 6.5.

6.6.3 SGI I/O Subsystem

To support the high computing power provided by multiple processors, in a real system, careful attention needs to be devoted to providing matching I/O capability. To provide scalable I/O performance, the SGI Challenge allows for multiple I/O cards to be placed on the system bus, each card providing a local 320 Mbytes/sec proprietary I/O bus. Personality ASICs are provided to act as an interface between the I/O bus and standards conforming (e.g., ethernet, VME, SCSI, HPPI) and non-standards conforming (e.g., SGI Graphics) devices. Figure 6-13 shows a block-level diagram of the SGI Challenge's PowerChannel-2 I/O subsystem.

As shown in the figure, the proprietary HIO input/output bus is at the core of the I/O subsystem. It is a 64-bit wide multiplexed address/data bus that runs off the same clock as the system bus. It supports split read-transactions, with up to four outstanding transactions per device. In contrast to the main system bus, it uses centralized arbitration as latency is much less of a concern. However, arbitration is pipelined so that bus bandwidth is not wasted. Furthermore, since the HIO bus supports several different transaction lengths (it does not require every transaction to handle a full cache block of data), at time of request transactions are required to indicate their length. The arbiter uses this information to ensure more efficient utilization of the bus. The narrower HIO bus allows the personality ASICs to be cheaper than if they were to directly interface to the very wide system bus. Also, common functionality needed to interface to the system bus can be shared by the multiple personality ASICs.

---

1. Note that on both the outgoing and return paths, the memory request passes through an asynchronous boundary. This adds a double synchronizer delay in both directions, about 30ns on average in each direction. The benefit of decoupling is that the CPU can run at a different clock rate than the system bus, thus allowing for migration to higher clock-rate CPUs in the future while keeping the same bus clock rate. The cost, of course, is the extra latency.

The newer generation of processor, the MIPS R10000, allows the processor to restart after only the needed word has arrived, without having to wait for the complete cache block to arrive. This early restart option reduces the miss latency.
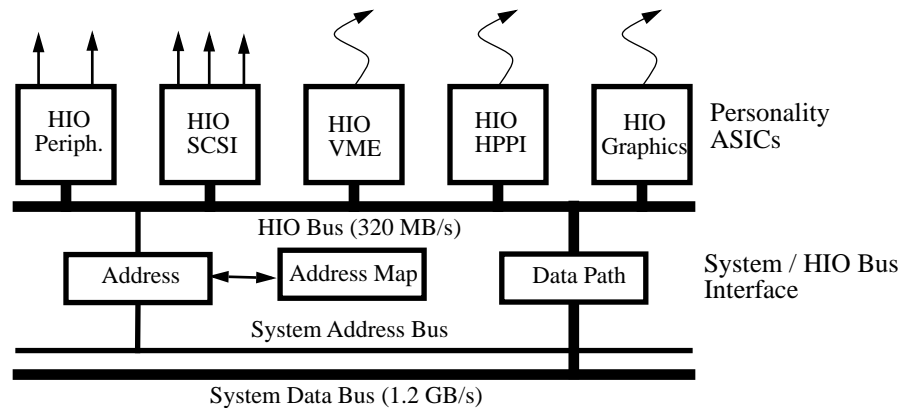
Figure 6-16 High-level organization of the SGI Challenge Powerchannel-2 I/O subsystem.

Each I/O board provides an interface to the Powerpath-2 system bus, and an internal 64-bits wide "HIO" I/O bus with peak bandwidth of 320 Mbytes/sec. The narrower HIO bus lowers the cost of interfacing to it, and it supports a number of personality ASICs which, in turn, support standard buses and peripherals.

HIO interface chips can request DMA read/write to system memory using the full 40-bit system address, make a request for address translation using the mapping resource in the system interface (e.g., for 32-bit VME), request interrupting the processor, or respond to processor I/O (PIO) reads. The system bus interface provides DMA read responses, results of address translation, and passes on PIO reads to devices.

To the rest of the system (processor boards and main-memory boards), the system bus interface on the I/O board provides a clean interface; it essentially acts like another processor board. Thus, when a DMA read makes it through the system-bus interface onto the system bus, it becomes a Powerpath-2 read, just like one that a processor would issue. Similarly, when a full-cache-block DMA write goes out, it becomes a special block write transaction on the bus that invalidates copies in all processors' caches. Note that even if a processor had the block dirty in its local cache, we do not want it to write it back, and hence the need for the special transaction.

To support partial block DMA writes, special care is needed, because data must be merged coherently into main memory. To support these partial DMA writes, the system-bus interface includes a fully associative, four block cache, that snoops on the Powerpath-2 bus in the usual fashion. The cache blocks can be in one of only two states: (i) invalid or (ii) modified exclusive. When a partial DMA write is first issued, the block is brought into this cache in modified exclusive state, invalidating its copy in all processors' caches. Subsequent partial DMA writes need not go to the Powerpath-2 bus if they hit in this cache, thus increasing the system bus efficiency. This modified exclusive block goes to the invalid state and supplies its contents on the system bus: (i) on any Powerpath-2 bus transaction accessing this block; (ii) when another partial DMA write causes this cache block to be replaced; and (iii) on any HIO bus read transaction that accesses this block. While DMA reads could have also used this four-block cache, the designers felt that partial DMA reads were rare, and the gains from such an optimization would have been minimal.

The mapping RAM in the system-bus interface provides general-purpose address translation for I/O devices. For example, it may be used to map small address spaces such as VME-24 or VME-32 into the 40-bit physical address space of the Powerpath-2 bus. Two types of mapping are sup-

ported: one level and two level. One-level mappings simply return one of the 8K entries in the mapping RAM, where by convention each entry maps 2 Mbytes of physical memory. In the two-level scheme, the map entry points to the page tables in main memory. However, each 4 KByte page has its own entry in the second-level table, so virtual pages can be arbitrarily mapped to physical pages. Note that PIOs face a similar translation problem, when going down to the I/O devices. Such translation is not done using the mapping RAM, but is directly handled by the personality ASIC interface chips.

The final issue that we explore for I/O is flow control. All requests proceeding from the I/O interfaces/devices to the Powerpath-2 system bus are implicitly flow controlled. For example, the HIO interface will not issue a read on the Powerpath-2 bus unless it has buffer space reserved for the response. Similarly, the HIO arbiter will not grant the bus to a requestor unless the system interface has room to accept the transaction. In the other direction, from the processors to I/O devices, however, PIOs can arrive unsolicited and they need to be explicitly flow controlled.

The flow control solution used in the Challenge system is to make the PIOs be solicited. After reset, HIO interface chips (e.g., HIO-VME, HIO-HPPI) signal their available PIO buffer space to the system-bus interface using special requests called IncPIO. The system-bus interface maintains this information in a separate counter for each HIO device. Every time a PIO is sent to a particular device, the corresponding count is decremented. Every time that device retires a PIO, it issues another IncPIO request to increment its counter. If the system bus interface receives a PIO for a device that has no buffer space available, it rejects (NACKs) that request on the Powerpath-2 bus and it must be retried later.

6.6.4 SGI Challenge Memory System Performance

The access time for various levels of the SGI Challenge memory system can be determined using the simple read microbenchmark from Chapter 3. Recall, the microbenchmark measures the average access time in reading elements of an array of a given size with a certain stride. Figure 6-17 shows the read access time for a range of sizes and strides. Each curve shows the average access time for a given size as a function of the stride. Arrays smaller than 32 KB fit entirely in the first level cache. Level two cache accesses have an access time of roughly 75 ns, and the inflection point shows that the transfer size is 16 bytes. The second bump shows the additional penalty of roughly 140 ns for a TLB miss, and the page size in 8 KB. With a 2 MB array accesses miss in the L2 cache, and we see that the combination of the L2 controller, Powerpath bus protocol and DRAM access result in an access time of roughly 1150 ns. The minimum bus protocol of 13 cycles at 50 MHz accounts for 260 ns of this time. TLB misses add roughly 200 ns. The simple ping-pong microbenchmark, in which a pair of nodes each spin on a flag until it indicates

their turn and then set the flag to signal the other shows a round-trip time of 6.2μs, a little less than four memory accesses.
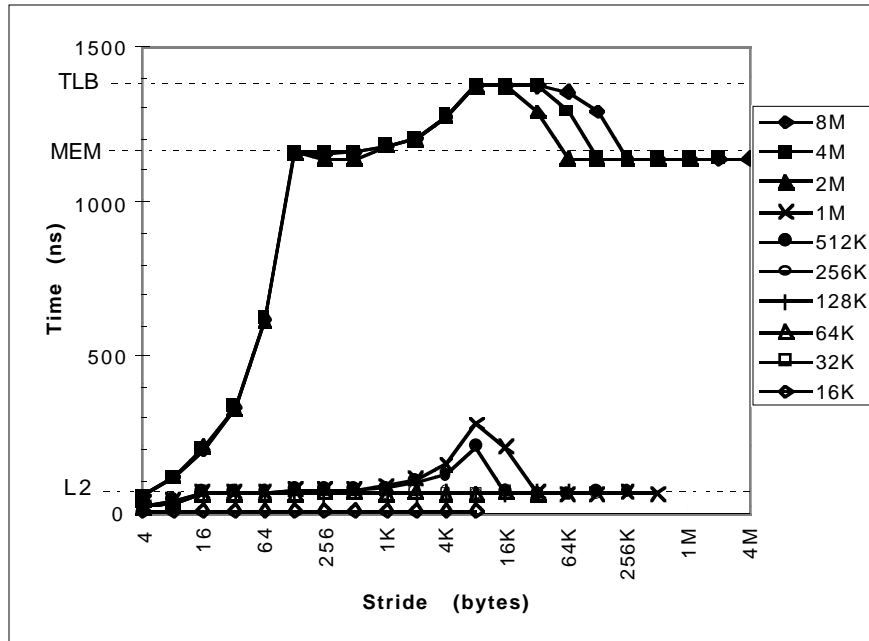
Figure 6-17 Read microbenchmark results for the SGI Challenge.

6.6.5 Sun Gigaplane System Bus

The Sun Gigaplane is a non-multiplexed, split-phase (or packet switched) bus with 256-bit data lines and 41-bit physical addresses, clocked at 83.5 MHz. It is a *centerplane design*, rather than a backplane, so cards plug into both sides of it. The total length of the bus is 18", so eight boards can plug into each side with 2" of cooling space between boards and 1" spacing between connectors. In sharp contrast to the SGI Challenge Powerpath-2 bus, the bus can support up to 112 outstanding transactions, including up to 7 from each board, so it is designed for devices that can sustain multiple outstanding transactions, such as lock-up free caches. The electrical and mechanical design allows for live insertion (hot plug) of processing and I/O modules.

Looking at the bus in more detail, it consists of 388 signals: 256 data, 32 ECC, 43 address (with parity), 7 id tag, 18 arbitration, and a number of configuration signals. The electrical design allows for turn-around with no dead cycles. Emphasis is placed on minimizing the latency of operations, and the protocol (illustrated in Figure 6-18) is quite different from that on the SGI Challenge. A novel collision-based arbitration technique is used to avoid the cost of bus arbitration. When a requestor arbitrates for the address bus, if the address bus is not scheduled to be in use from the previous cycle, it speculatively drives its request on the address bus. If there are no other requestors in that cycle, it wins arbitration and has already passed the address, so it can continue with the remainder of the transaction. If there is an address collision, the requestor that wins arbitration simply drives the address again in the next cycle, as it would with conventional arbitration. The 7-bit tag associated with the request is presented in the following cycle. The snoop state is associated with the address phase, not the data phase. Five cycles after the address, all

boards assert their snoop signals (shared, owned, mapped, and ignore). In the meantime, the board responsible for the memory address (the home board) can request the data bus three cycles after the address, before the snoop result. The DRAM access can be started speculatively, as well. When home board wins arbitration, it must assert the tag two cycles later, informing all devices of the approaching data transfer. Three cycles after driving the tag and two cycles before the data, the home board drives a status signal, which will indicate that the data transfer is cancelled if some cache owns the block (as detected in the snoop state). The owner places the data on the bus by arbitrating for the data bus, driving the tag, and driving the data. Figure 6-18 shows a second read transaction, which experiences a collision in arbitration, so the address is supplied in the conventional slot, and cache ownership, so the home board cancels its data transfer.

Like the SGI Challenge, invalidations are ordered by the BusRdX transactions on the address bus and handled in FIFO fashion by the cache subsystems, thus no acknowledgment of invalidation completion is required. To maintain sequential consistency, is still necessary to gain arbitration for the address bus before allowing the writing processor to proceed with memory operations past the write[1].
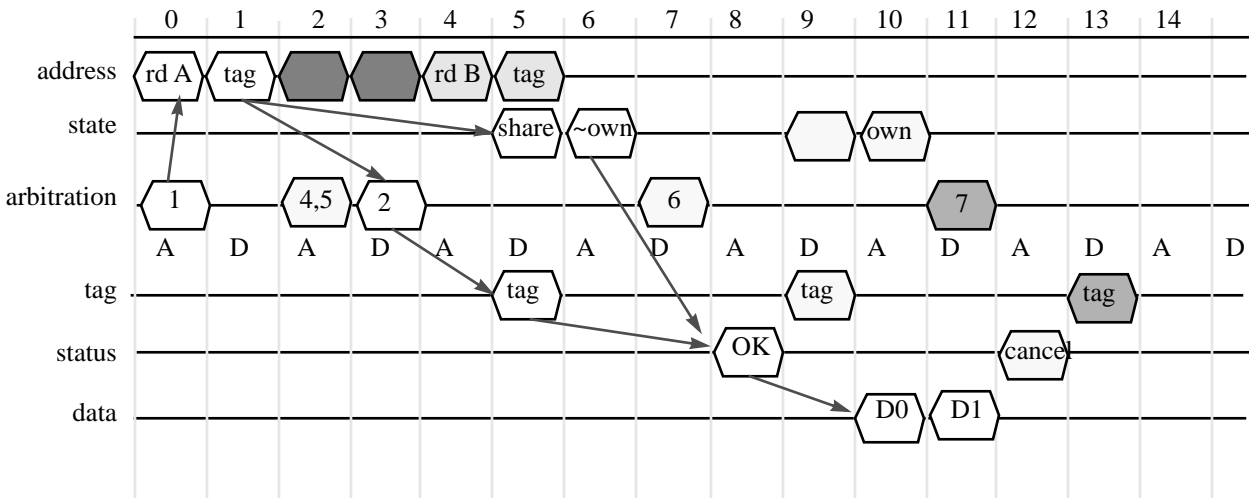
Figure 6-18 Sun Gigaplane Signal Timing for BusRd with fast address arbitration

Board 1 initiates a read transaction with fast arbitration, which is responded to by home board 2. Boards 4 and 5 collide during arbitration, board 4 wins, and initiates a read transaction. Home board 6 arbitrates for the data bus and then cancels its response. Eventually the owning cache, board 7, responds with the data. Board 5 retry is not shown.

6.6.6 Sun Processor and Memory Subsystem

In the Sun Enterprise, each processing board has two processors, each with external L2 caches, and two banks of memory connected through a cross-bar, as shown in Figure 6-19. Data lines within the UltraSPARC module are buffered to drive an internal bus, called the UPA (universal port architecture) with an internal bandwidth of 1.3 GB/s. A very wide path to memory is pro-

---

1. The SPARC V9 specification weakens the consistency model in this respect to allow the processor to employ write buffers, which we discuss in more depth in Chapter 6.

vided so that a full 64 byte cache block can be read in a singe memory cycle, which is two bus cycles in length. The address controller adapts the UPA protocol to the Gigaplane protocol, realizes the cache coherency protocol, provides buffering, and tracks the potentially large number of outstanding transactions. It maintains a set of duplicate tags (state and address, but no data) for the L2 cache. Although the UltraSPARC implements a 5-state MOESI protocol, the D-tags maintain an approximation to the state: owned, shared, invalid. It essentially combines states which are handled identically at the Gigaplane level. In particular, it needs to know if the L2 cache has a block and if that block is the only block in a cache. It does not need to know if that block is clean or dirty. For example, on a BusRd the block will need to be flushed onto the bus if it is in the L2 cache as modified, owned (flushed since last modified), or exclusive (not shared when read and not modified since), thus the D-tags represent only 'owned'. This has the advantage that the address controller need not be informed when the UltraSPARC elevates a block from exclusive to modified. It will be informed of a transition from invalid, shared, or owned to modified, because it needs to initiate bus transaction.
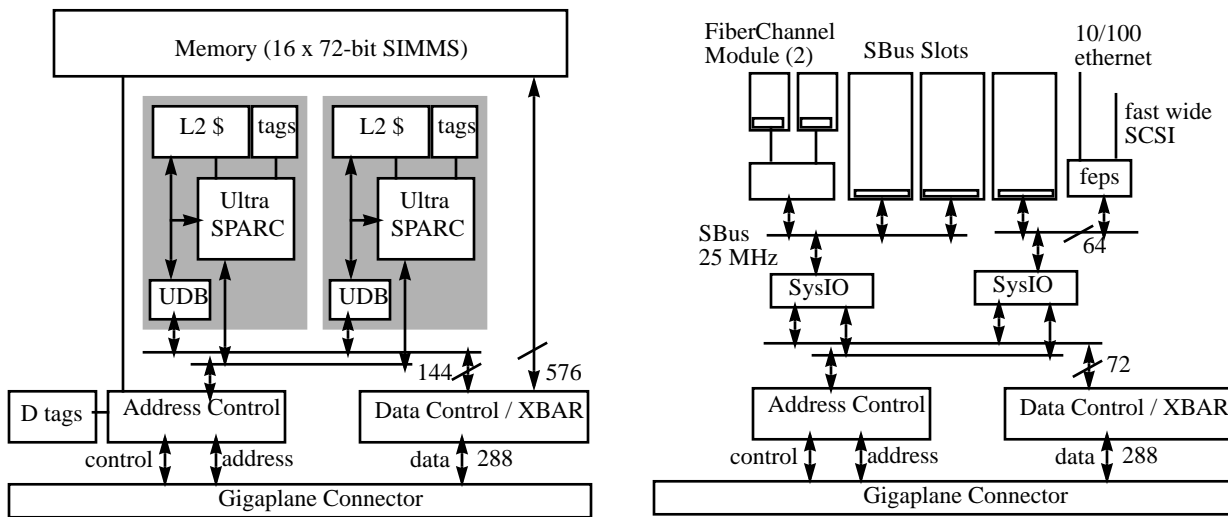
Figure 6-19 Organization of the Sun Enterprise Processing and I/O Board

Processing board contains two UltraSPARC modules with L2 caches on an internal bus, and two wide memory banks interfaced to the system bus through two ASICs. The Address Controller adapts between the two bus protocols and implements the cache coherency protocol. The Data Controller is essentially a cross bar. The I/O board uses the same two ASICs to interface to two I/O controllers. The SysIO asics essentially appear to the bus as a one block cache. On the other side, they support independent I/O busses and interfaces to FiberChannel, ethernet, and SCSI.

6.6.7 Sun I/O Subsystem

The Enterprise I/O board uses the same bus interface ASICS as the processing board. The internal bus is only half as wide and there is no memory path. The I/O boards only do cache block transactions, just like the processing boards, in order to simplify the design of the main bus. The SysI/O ASICs implement a single block cache on behalf of the I/O devices. Two independent 64-bit 25 MHz SBus are supported. One of these supports two dedicated FiberChannel modules providing a redundant, high bandwidth interconnect to large disk storage arrays. The other provides dedicated ethernet and fast wide SCSI connections. In addition, three SBUS interface cards can be plugged into the two busses to support arbitrary peripherals, including a 622 Mb/s ATM inter-

face. The I/O bandwidth, the connectivity to peripherals, and the cost of the I/O subsystem scales with the number of I/O cards.

6.6.8 Sun Enterprise Memory System Performance

The access time for various level of the Sun Enterprise via the read microbenchmark is shown in Figure 6-20. Arrays of 16 KB or less fit entirely in the first level cache. Level two cache accesses have an access time of roughly 40 ns, and the inflection point shows that the transfer size is 16 bytes. With a 1 MB array accesses miss the L2 cache, and we see that the combination of the L2 controller, bus protocol and DRAM access result in an access time of roughly 300 ns. The minimum bus protocol of 11 cycles at 83.5 MHz accounts for 130 ns of this time. TLB misses add roughly 340 ns. The machine has a software TLB handler. The simple ping-pong microbenchmark, in which a pair of nodes each spin on a flag until it indicates their turn and then set the flag to signal the other shows a round-trip time of 1.7 μs, roughly five memory accesses.
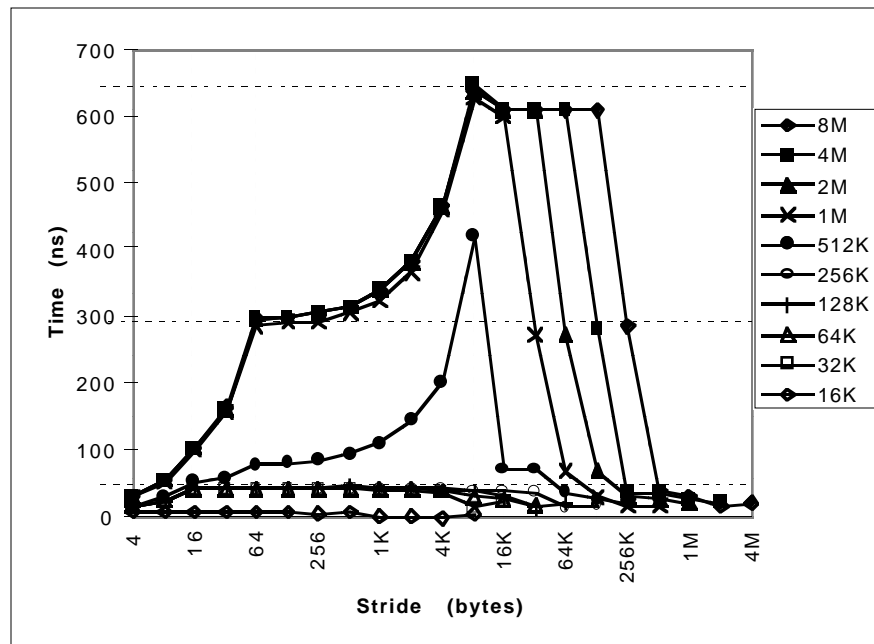
Figure 6-20 Read Microbenchmark results for the Sun Enterprise.

6.6.9 Application Performance

Having understood the machines and their microbenchmark performance, let us examine the performance obtained on our parallel applications. Absolute performance for commercial machines is not presented in this book; instead, the focus is on performance improvements due to parallelism. Let us first look at application speedups and then at scaling, using only the SGI Challenge for illustration.

Application Speedups

Figure 6-21 shows the speedups obtained on our six parallel programs, for two data set sizes each. We can see that the speedups are quite good for most of the programs, with the exception of the Radix sorting kernel. Examining the breakdown of execution time for the sorting kernel shows that the vast majority of the time is spent stalled on data access. The shared bus simply gets swamped with the data and coherence traffic due to the permutation phase of the sort, and the resulting contention destroys performance. The contention also leads to severe load imbalances in data access time and hence time spent waiting at global barriers. It is unfortunately not alleviated much by increasing the problem size, since the communication to computation ratio in the permutation phase is independent of problem size. The results shown are for a radix value of 256, which delivers the best performance over the range of processor counts for both problem sizes. Barnes-Hut, Raytrace and Radiosity speed up very well even for the relatively small input problems used. LU does too, and the bottleneck for the smaller problem at 16 processors is primarily load imbalance as the factorization proceeds along the matrix. Finally, the bottleneck for the small Ocean problem size is both the high communication to computation ratio and the imbalance this generates since some partitions have fewer neighbors than others. Both problems are alleviated by running larger data sets.
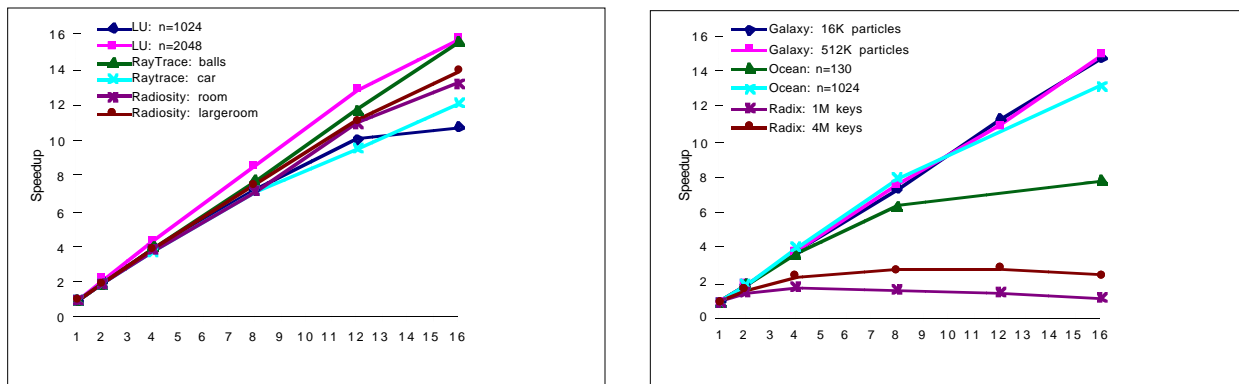
Figure 6-21 Speedups for the parallel applications on the SGI Challenge.

The block size for the blocked LU factorization is 32-by-32.

Scaling

Let us now examine the impact of scaling for a few of the programs. Following the discussion of Chapter 4, we look at the speedups under the different scaling models, as well as at how the work done and the data set size used change. Figure 6-22 shows the results for the Barnes-Hut and Ocean applications. "Naive" time-constrained (TC) or memory-constrained (MC) refers to scaling on the number of particles or grid length (n) without changing the other application parameters (accuracy or the number of time steps). It is clear that the work done under realistic MC scaling grows much faster than linearly in the number of processors in both applications, so the parallel execution time grows very quickly. The number of particles or the grid size that can be simulated under TC scaling grows much more slowly than under MC scaling, and also much more slowly than under naive TC where n is the only application parameter scaled. Scaling the

other application parameters causes the work done and execution time to increase, leaving much less room to grow n. Data set size is controlled primarily by n, which explains its scaling trends.

The speedups under different scaling models are measured as described in Chapter 4. Consider the Barnes-Hut galaxy simulation, where the speedups are quite good for this size of machine under all scaling models. The differences can be explained by examining the major performance factors. The communication to computation ratio in the force calculation phase depends primarily on the number of particles. Another important factor that affects performance is the relative amount of work done in the force calculation phase, which speeds up well, to the amount done in the tree-building phase which does not. This tends to increase with greater accuracy in force computation, i.e. smaller θ. However, smaller θ (and to a lesser extent greater n) increase the working set size [SHG93], so the scaled problem that changes θ may have worse first-level cache behavior than the baseline problem (with a smaller n and larger θ) on a uniprocessor. These factors can be used to explain why naive TC scaling yields better speedups than realistic TC scaling. The working sets behave better, and the communication to computation ratio is more favorable since n grows more quickly when θ and Δt are not scaled.

The speedups for Ocean are quite different under different models. Here too, the major controlling factors are the communication to computation ratio, the working set size, and the time spent in different phases. However, all the effects are much more strongly dependent on the grid size relative to number of processors. Under MC scaling, the communication to computation ratio does not change with the number of processors used, so we might expect the best speedups. However, as we scale two effects become visible. First, conflicts across grids in the cache increase as a processor's partitions of the grids become further apart in the address space. Second, more time is spent in the higher levels of the multigrid hierarchy in the solver, which have worse parallel performance. The latter effect turns out to be alleviated when accuracy and time-step interval are refined as well, so realistic MC scales a little better than naive MC. Under naive TC scaling, the growth in grid size is not fast enough to cause major conflict problems, but good enough that communication to computation ratio diminishes significantly, so speedups are very good. Realistic TC scaling has a slower growth of grid size and hence improvement in communication to computation ratio, and hence lower speedups. Clearly, many effects play an important role in performance under scaling, and which scaling model is most appropriate for an application affects the results of evaluating a machine.

6.7 Extending Cache Coherence

The techniques for achieving cache coherence extend in many directions. This examines a few important directions: scaling down with shared caches, scaling in functionality with virtually indexed caches and translation lookaside buffers (TLBs), and scaling up with non-bus interconnects.

6.7.1 Shared-Cache Designs

Grouping processors together to share a level of the memory hierarchy (e.g., the first or the second-level cache) is a potentially attractive option for shared-memory multiprocessors, especially as we consider designs with multiple processors on a chip. Compared with each processor having its own memory at that level of the hierarchy, it has several potential benefits. The benefits—like
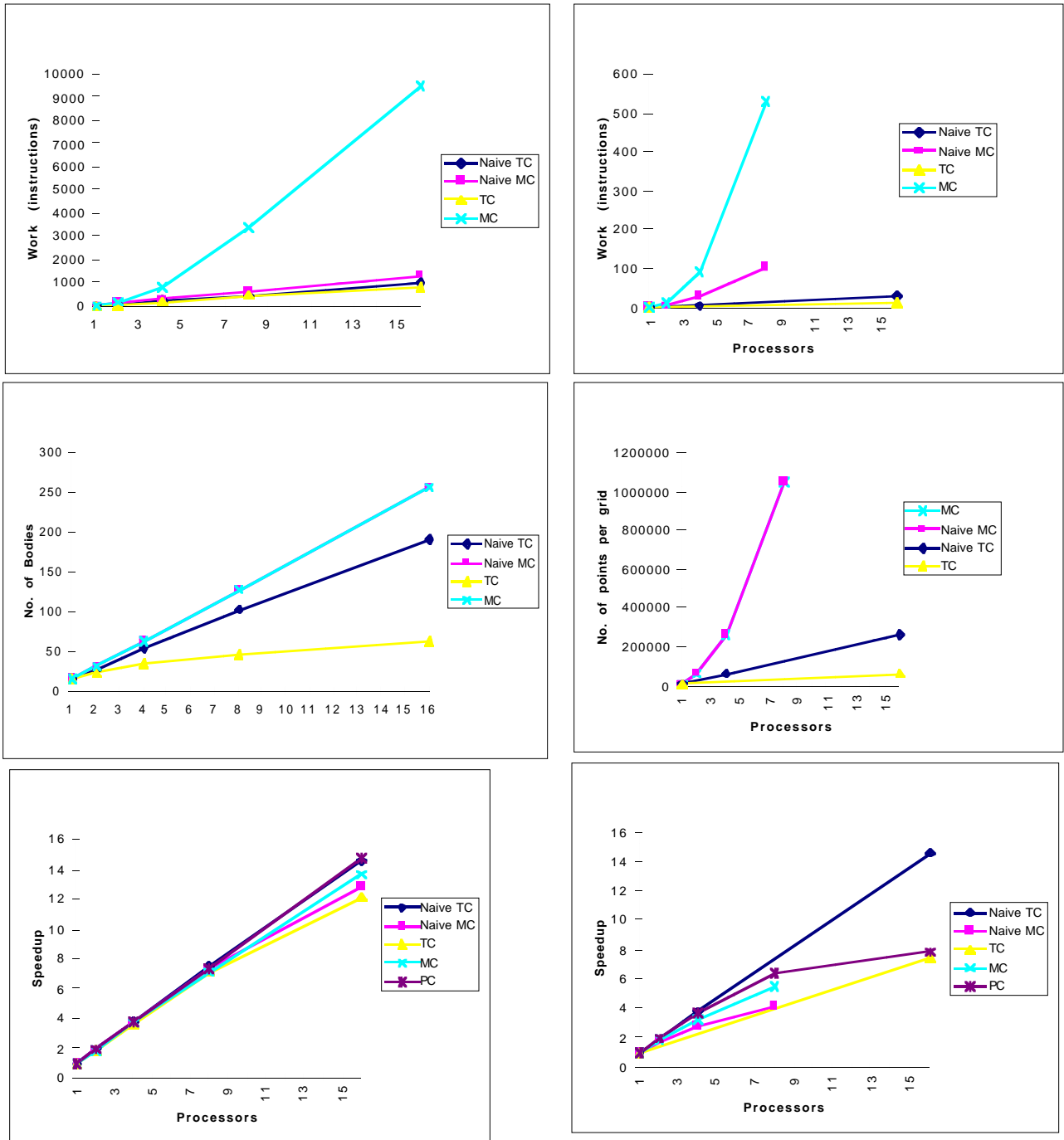
Figure 6-22 Scaling results for Barnes-Hut (left) and Ocean (right) on the SGI Challenge.

The graphs show the scaling of work done, data set size, and speedups under different scaling models. PC, TC, and MC refer to problem constrained, time constrained and memory constrained scaling, respectively. The top set of graphs shows that the work needed to solve the problem grows very quickly under realistic MC scaling for both applications. The middle set of graphs shows that the data set size that can be run grows much more quickly under MC or naive TC scaling than under realistic TC scaling. The impact of scaling model on speedup is much larger for Ocean than for Barnes-Hut, primarily because the communication to computation ratio is much more strongly dependent on problem size and number of processors in Ocean.

the later drawbacks—are encountered when sharing at any level of the hierarchy, but are most extreme when it is the first-level cache that is shared among processors. The benefits of sharing a cache are:

- It eliminates the need for cache-coherence at this level. In particular, if the first-level cache is shared then there are no multiple copies of a cache block and hence no coherence problem whatsoever.

- It reduces the latency of communication that is satisfied within the group. The latency of communication between processors is closely related to the level in the memory hierarchy where they meet. When sharing the first-level cache, communication latency can be as low as 2-10 clock cycles. The corresponding latency when processors meet at the main-memory level is usually many times larger (see the Challenge and Enterprise case studies). The reduced latency enables finer-grained sharing of data between tasks running on the different processors.

- Once one processor misses on a piece of data and brings it into the shared cache, other processors in the group that need the data may find it already there and will not have to miss on it at that level. This is called prefetching data across processors. With private caches each processor would have to incur a miss separately. The reduced number of misses reduces the bandwidth requirements at the next level of the memory and interconnect hierarchy.

- It allows more effective use of long cache blocks. Spatial locality is exploited even when different words on a cache block are accessed by different processors in a group. Also, since there is no cache coherence within a group at this level there is also no false sharing. For example, consider a case where two processors P1 and P2 read and write every alternate word of a large array, and think about the differences when they share a first-level cache and when they have private first-level caches.

- The working sets (code or data) of the processors in a group may overlap significantly, allowing the size of shared cache needed to be smaller than the combined size of the private caches if each had to hold its processor's entire working set.

- It increases the utilization of the cache hardware. The shared cache does not sit idle because one processor is stalled, but rather services other references from other processors in the group.

- The grouping allows us to effectively use emerging packaging technologies, such as multi-chip-modules, to achieve higher computational densities (computation power per unit area).

The extreme form of cache sharing is the case in which all processors share a first level cache, below which is a shared main memory subsystem. This completely eliminates the cache coherence problem. Processors are connected to the shared cache by a switch. The switch could even be a bus but is more likely a crossbar to allow cache accesses from different processors to proceed in parallel. Similarly, to support the high bandwidth imposed by multiple processors, both the cache and the main memory system are interleaved.

An early example of a shared-cache architecture is the Alliant FX-8 machine, designed in the early 1980s. An Alliant FX-8 contained up to 8 custom processors. Each processor was a pipelined implementation of the 68020 instruction set, augmented with vector instructions, and had a clock cycle of 170ns. The processors were connected using a crossbar to a 512Kbyte, 4-way interleaved cache. The cache had 32byte blocks, and was writeback, direct-mapped, and lock-up free allowing each processor to have two outstanding misses. The cache bandwidth was eight 64-bit words per instruction cycle. The interleaved main memory subsystem had a peak bandwidth of 192 Mbytes/sec.
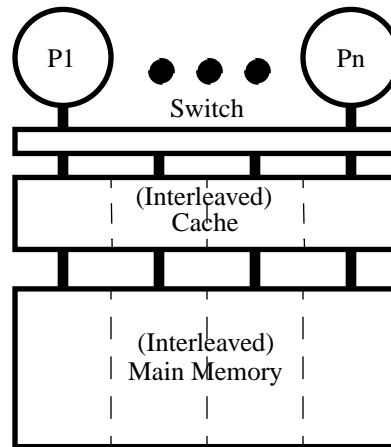
Figure 6-23 Generic architecture for a shared-cache multiprocessor.

The interconnect is placed between the processors and the first-level cache. Both the cache and the memory system may be interleaved to provide high bandwidth.

A somewhat different early use of the shared-cache approach was exemplified by the Encore Multimax, a contemporary of the FX-8. The Multimax was a snoopy cache-coherent multiprocessor, but each private cache supported two processors instead of one (with no need for coherence within a pair). The motivation for Encore at the time was to lower the cost of snooping hardware, and to increase the utilization of the cache given the very slow, multiple-CPI processors.

Today, shared first-level caches are being investigated for single-chip multiprocessors, in which four-to-eight multiprocessors share an on-chip first-level cache. These can be used in themselves as multiprocessors, or as the building-blocks for larger systems that maintain coherence among the single-chip shared-cache groups. As technology advances and the number of transistors on a chip reaches several tens or hundreds of millions, this approach becomes increasingly attractive. Workstations using such chips will be able to offer very high-performance for workloads requiring either fine-grain or coarse-grain parallelism. The question is whether this is a more effective approach or one that uses the hardware resources to build more complex processors.

However, sharing caches, particularly at first level, has several disadvantages and challenges:

- The shared cache has to satisfy the bandwidth requirements from multiple processors, restricting the size of a group. The problem is particularly acute for shared first-level caches, which are therefore limited to very small numbers of processors. Providing the bandwidth needed is one of the biggest challenges of the single-chip multiprocessor approach.

- The hit latency to a shared cache is usually higher than to a private cache at the same level, due to the interconnect in between. This too is most acute for shared first-level caches, where the imposition of a switch between the processor and the first-level cache means that either the machine clock cycle is elongated or that additional delay-slots are added for load instructions in the processor pipeline. The slow down due to the former is obvious. While compilers have some capability to schedule independent instructions in load delay slots, the success depends on the application. Particularly for programs that don't have a lot of instruction-level

parallelism, some slow down is inevitable. The increased hit latency is aggravated by contention at the shared cache, and correspondingly the miss latency is also increased by sharing.

- For the above reasons, the design complexity for building an effective system is higher.

- Although a shared cache need not be as large as the sum of the private caches it replaces, it is still much larger and hence slower than an individual private cache. For first-level caches, this too will either elongate the machine clock cycle or lead to multiple processor-cycle cache access times.

- The converse of overlapping working sets (or constructive interference) is the performance of the shared cache being hurt due to cache conflicts across processor reference streams (destructive interference). When a shared-cache multiprocessor is used to run workloads with little data sharing, for example a parallel compilation or a database/transaction processing workload, the interference in the cache between the data sets needed by the different processors can hurt performance substantially. In scientific computing where performance is paramount, many programs try to manage their use of the per-processor cache very carefully, so that the many arrays they access do not interfere in the cache. All this effort by the programmer or compiler can easily be undone in a shared-cache system.

- Finally, shared caches today do not meet the trend toward using commodity microprocessor technology to build cost-effective parallel machines, particularly shared first-level caches.

Since many microprocessors already provide snooping support for first-level caches, an attractive approach may be to have private first-level caches and a shared second-level cache among groups of processors. This will soften both the benefits and drawbacks of shared first-level caches, but may be a good tradeoff overall. The shared cache will likely be large to reduce destructive interference. In practice, packaging considerations will also have a very large impact on decisions to share caches.

6.7.2 Coherence for Virtually Indexed Caches

Recall from uniprocessor architecture the tradeoffs between physically and virtually indexed caches. With physically indexed first-level caches, for cache indexing to proceed in parallel with address translation requires that the cache be either very small or very highly associative, so the bits that do not change under translation ($log_2(page\_size)$ bits or a few more if page coloring is used) are sufficient to index into it [HeP90]. As on-chip first-level caches become larger, virtually indexed caches become more attractive. However, these have their own, familiar problems. First, different processors may use the same virtual address to refer to unrelated data in different address spaces. This can be handled by flushing the whole cache on a context switch or by associating address space identifier (ASID) tags with cache blocks in addition to virtual address tags. The more serious problem for cache coherence is synonyms: distinct virtual pages, from the same or different processes, pointing to the same physical page for sharing purposes. With virtually addressed caches, the same physical memory block can be fetched into two distinct blocks at different indices in the cache. As we know, this is a problem for uniprocessors, but the problem extends to cache coherence in multiprocessors as well. If one processor writes the block using one virtual address synonym and another reads it using a different synonym, then by simply putting virtual addresses on the bus and snooping them the write to the shared physical page will not become visible to the latter processor. Putting virtual addresses on the bus also has another drawback, requiring I/O devices and memory to do virtual to physical translation since they deal with physical addresses. However, putting physical addresses on the bus seems to require reverse

translation to look up the caches during a snoop, and this does not solve the synonym coherence problem anyway.

There are two main software solutions to avoiding the synonym problem: forcing synonyms to be the same in the bits used to index the cache if these are more than $log_2(page\_size)$ (i.e. forcing them to have the same page color), and forcing processes to use the same shared virtual address when referring to the same page (as in the SPUR research project [HEL+86]).

Sophisticated cache designs have also been proposed to solve the synonym coherence problem in hardware [Goo87]. The idea is to use virtual addresses to look up the cache on processor accesses, and to put physical addresses on the bus for other caches and devices to snoop. This requires mechanisms to be provided for the following: (i) if a lookup with the virtual address fails, then to look up the cache with the physical address (which is by now available) as well in case it was brought in by a synonym access, (ii) to ensure that the same physical block is never in the same cache under two different virtual addresses at the same time, and (iii) to convert a snooped physical address to a effective virtual address to look up the snooping cache. One way to accomplish these goals is for caches to maintain both virtual and physical tags (and states) for their cached blocks, indexed by virtual and physical addresses respectively, and for the two tags for a block to point to each other (i.e. to store the corresponding physical and virtual indices, respectively, see Figure 6-24). The cache data array itself is indexed using the virtual index (or the pointer from the physical tag entry, which is the same). Let's see at a high level how this provides the above mechanisms.
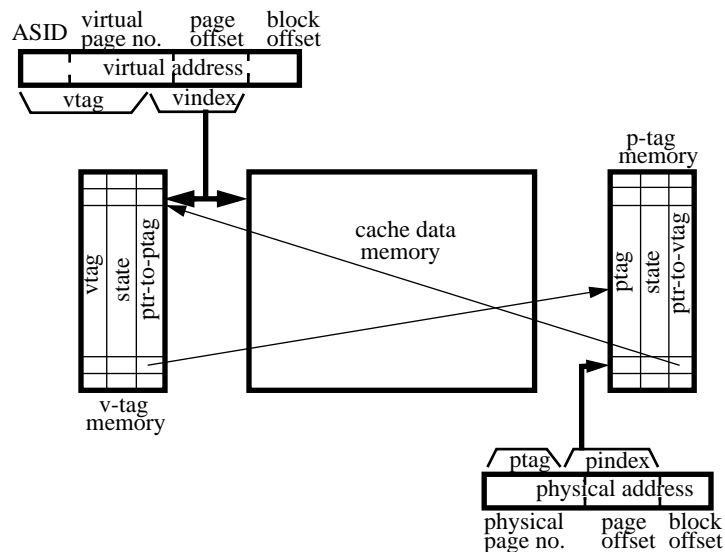
Figure 6-24 Organization of a dual-tagged virtually-addressed cache.

The v-tag memory on the left services the CPU and is indexed by virtual addresses. The p-tag memory on the right is used for bus snooping, and is indexed by physical addresses. The contents of the memory block are stored based on the index of the v-tag. Corresponding p-tag and v-tag entries point to each other for handling updates to cache.

A processor looks up the cache with its virtual address, and at the same time the virtual to physical translation is done by the memory management unit in case it is needed. If the lookup with the virtual address succeeds, all is well. If it fails, the translated physical address is used to look

up the physical tags, and if this hits the block is found through the pointer in the physical tag. This achieves the first goal. A virtual miss but physical hit detects the possibility of a synonym, since the physical block may have been brought in via a different virtual address. In a direct-mapped cache, it must have, and let us assume a direct-mapped cache for concreteness. The pointer contained in the physical tags now points to a different block in the cache array (the synonym virtual index) than does the virtual index of the current access. We need to make the current virtual index point to this physical data, and reconcile the virtual and physical tags to remove the synonym. The physical block, which is currently pointed to by the synonym virtual index, is copied over to replace the block pointed to by the current virtual index (which is written back if necessary), so references to the current virtual index will hereafter hit right away. The former cache block is rendered invalid or inaccessible, so the block is now accessible only through the current virtual index (or through the physical address via the pointer in the physical tag), but not through the synonym. A subsequent access to the synonym will miss on its virtual address lookup and will have to go through this procedure. Thus, a given physical block is valid only in one (virtually indexed) location in the cache at any given time, accomplishing the second goal. Note that if both the virtual and physical address lookups fail (a true cache miss), we may need up to two writebacks. The new block brought into the cache will be placed at the index determined from the virtual (not physical) address, and the virtual and physical tags and states will be suitably updated to point to each other.

The address put on the bus, if necessary, is always a physical address, whether for a writeback, a read miss, or a read-exclusive or upgrade. Snooping with physical addresses from the bus is easy. Explicit reverse translation is not required, since the information needed is already there. The physical tags are looked up to check for the presence of the block, and the data are found from the pointer (corresponding virtual index) it contains. If action must be taken, the state in the virtual tag pointed to by the physical tag entry is updated as well. Further details of how such a cache system operates can be found in [Goo87]. This approach has also been extended to multi-level caches, where it is even more attractive: the L1 cache is virtually-tagged to speed cache access, while the L2 cache is physically tagged [WBL89].

6.7.3 Translation Lookaside Buffer Coherence

A processor's *translation lookaside buffer* (TLB) is a cache on the page table entries (PTEs) used for virtual to physical address translation. A PTE can come to reside in the TLBs of multiple processors, due to actual sharing or process migration. PTEs may be modified—for example when the page is swapped out or its protection is changed—leading to direct analog of the cache coherence problem.

A variety of solutions have been used for TLB coherence. Software solutions, through the operating system, are popular, since TLB coherence operations are much less frequent than cache coherence operations. The exact solutions used depend on whether PTEs are loaded into the TLB directly by hardware or through software, and on several other of the many variables in how TLBs and operating systems are implemented. Hardware solutions are also used by some systems, particularly when TLB operations are not visible to software. This section provides a brief overview of four approaches to TLB coherence: virtually addressed caches, software TLB shootdown, address space identifiers (ASIDs), and hardware TLB coherence. Further details can be found in [TBJ+99,Ros89,Tel90] and the papers referenced therein.

TLBs, and hence the TLB coherence problem, can be avoided by using virtually addressed caches. Address translation is now needed only on cache misses, so particularly if the cache miss rate is small we can use the page tables directly. Page table entries are brought into the regular data cache when they are accessed, and are therefore kept coherent by the cache coherence mechanism. However, when a physical page is swapped out or its protection changed, this is not visible to the cache coherence hardware, so they must be flushed from the virtually addressed caches of all processors by the operating system. Also, the coherence problem for virtually addressed caches must be solved. This approach was explored in the SPUR research project [HEL+86, WEG+86].

A second approach is called TLB shootdown. There are many variants that rely on different (but small) amounts of hardware support, usually including support for interprocessor interrupts and invalidation of TLB entries. The TLB coherence procedure is invoked on a processor, called the initiator, when it makes changes to PTEs that may be cached by other TLBs. Since changes to PTEs must be made by the operating system, it knows which PTEs are being changed and which other processors might be caching them in their TLBs (conservatively, since entries may have been replaced). The OS kernel locks the PTEs being changed (or the relevant page table sections, depending on the granularity of locking), and sends interrupts to other processors that it thinks have copies. The recipients disable interrupts, look at the list of page-table entries being modified (which is in shared memory) and locally invalidate those entries from their TLB. The initiator waits for them to finish, perhaps by polling shared memory locations, and then unlocks the page table sections. A different, somewhat more complex shootdown algorithm is used in the Mach operating system [BRG+89].

Some processor families, most notably the MIPS family from Silicon Graphics, used software-loaded rather than hardware-loaded TLBs, which means that the OS is involved not only in PTE modifications but also in loading a PTE into the TLB on a miss [Mip91]. In these cases, the coherence problem for process-private pages due to process migration can be solved using a third approach, that of ASIDs, which avoids interrupts and TLB shootdown. Every TLB entry has an ASID field associated with it, used just like in virtually addressed caches to avoid flushing the entire cache on a context switch. ASIDs here are like tags allocated dynamically by the OS, using a free pool to which they are returned when TLB entries are replaced; they are not associated with processes for their lifetime. One way to use the ASIDs, used in the Irix 5.2 operating system, is as follows. The OS maintains an array for each process that tracks the ASID assigned to that process on each of the processors in the system. When a process modifies a PTE, the ASID of that process for all other processors is set to zero. This ensures that when the process is migrated to another processor, it will find its ASID to be zero there so and the kernel will allocate it a new one, thus preventing use of stale TLB entries. TLB coherence for pages truly shared by processes is performed using TLB shootdown.

Finally, some processor families provide hardware instructions to invalidate other processors' TLBs. In the PowerPC family [WeS94] the "TLB invalidate entry" (`tlbie`) instruction broadcasts the page address on the bus, so that the snooping hardware on other processors can automatically invalidate the corresponding TLB entries without interrupting the processor. The algorithm for handling changes to PTEs is simple: the operating system first makes changes to the page table, and then issues a `tlbie` instruction for the changed PTEs. If the TLB is not software-loaded (it is not in the PowerPC) then the OS does not know which other TLBs might be caching the PTE so the invalidation must be broadcast to all processors. Broadcast is well-suited to a bus, but undesirable for the more scalable systems with distributed networks that will be discussed in subsequent chapters.

6.7.4 Cache coherence on Rings

Since the scale of bus-based cache coherent multiprocessors is fundamentally limited by the bus, it is natural to ask how it could be extended to other less limited interconnects. One straightforward extension of a bus is a ring. Instead of a single set of wires onto which all modules are attached, each module is attached to two neighboring modules. A ring is an interesting interconnection network from the perspective of coherence, since it inherently supports broadcast-based communication. A transaction from one node to another traverses link by link down the ring, and since the average distance of the destination node is half the length of the ring, it is simple and natural to let the acknowledgment simply propagate around the rest of the ring and return to the sender. In fact, the natural way to structure the communication in hardware is to have the sender place the transaction on the ring, and other nodes inspect (snoop) it as it goes by to see if it is relevant to them. Given this broadcast and snooping infrastructure, we can provide snoopy cache coherence on a ring even with physically distributed memory. The ring is a bit more complicated than a bus, since multiple transactions may be in progress around the ring simultaneously, and the modules see the transactions at different times and potentially in different order.

The potential advantage of rings over busses is that the short, point-to-point nature of the links allows them to be driven at very high clock rates. For example, the IEEE Scalable Coherent Interface (SCI) [Gus92] transport standard is based on 500 MHz 16-bit wide point-to-point links. The linear, point-to-point nature also allows the links to be extensively pipelined, that is, new bits can be pumped onto the wire by the source before the previous bits have reached the destination. This latter feature allows the links to be made long without affecting their throughput. A disadvantage of rings is that the communication latency is high, typically higher than that of buses, and grows linearly with the number of processors in the ring (on average, $p/2$ hops need to be traversed before getting to the destination on a unidirectional ring, and half that on a bidirectional ring).

Since rings are a broadcast media snooping cache coherence protocols can be implemented quite naturally on them. An early ring-based snoopy cache-coherent machine was the KSR1 sold by Kendall Square Research. [FBR93]. More recent commercial offerings use rings as the second level interconnect to connect together multiprocessor nodes, such as the Sequent NUMA-Q and Convex's Exemplar family [Con93,TSS+96]. (Both of these systems use a "directory protocol" rather than snooping on the ring interconnect, so we will defer discussion of them until ** Chapter 9**, when these protocols are introduced. Also, in the Exemplar, the interconnect within a node is not a bus or a ring, but a richly-connected, low-latency crossbar, as discussed below.) The University of Toronto's Hector system [VSL+91, FVS92] is a ring-based research prototype.

Figure 6-25 illustrates the organization of a ring-connected multiprocessor. Typically rings are used with physically distributed memory, but the memory may still be logically shared. Each node consists of a processor, its private cache, a portion of the global main memory, and a ring interface. The interface to the ring consists of an input link from the ring, a set of latches organized as a FIFO, and an output link to the ring. At each ring clock cycle the contents of the latches are shifted forward, so the whole ring acts as a circular pipeline. The main function of the latches is to hold a passing transaction long enough so that the ring-interface can decide whether to forward the message to the next node or not. A transaction may be taken out of the ring by storing the contents of the latch in local buffer memory and writing an empty-slot indicator into that latch instead. If a node wants to put something on the ring, it waits for a an opportunity to fill a passing empty slot and fills it. Of course, it is desirable to minimize the number of latches in each interface, to reduce the latency of transactions going around the ring.

Ring-Based Multiprocessor

Individual Node Structure
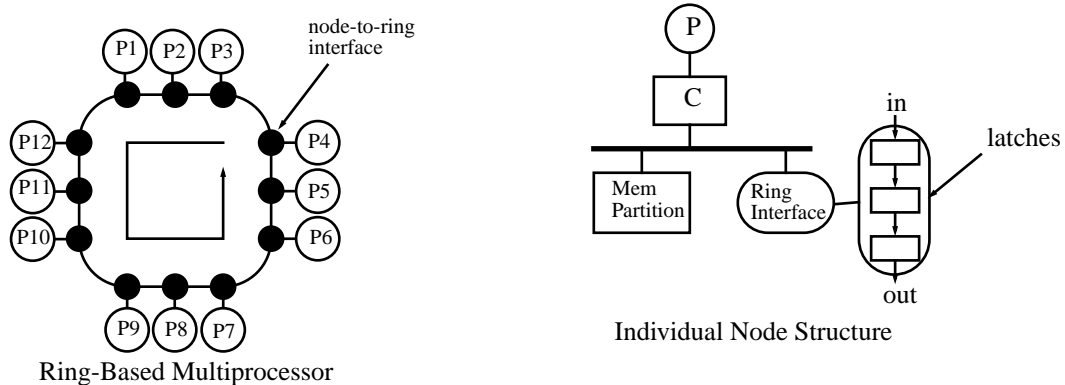
Figure 6-25 Organization of a single-ring multiprocessor.

The mechanism that determines when a node can insert a transaction on the ring, called the ring access control mechanism, is complicated by the fact that the data path of the ring is usually much narrower than size of the transactions being transferred on it. As a result, transactions need multiple consecutive slots on the ring. Furthermore, transactions (messages) on the ring can themselves have different sizes. For example, *request* messages are short and contain only the command and address, while *data reply* messages contain the contents of the complete memory block and are longer. The final complicating factor is that arbitration for access to the ring must be done in a distributed manner, since unlike in a bus there are no global wires.

Three main options have been used for access control (i.e., arbitration): token-passing rings, register-insertion rings, and slotted rings. In *token-passing rings* a special bit pattern, called a token, is passed around the ring, and only the node currently possessing it is allowed to transmit on the ring. Arbitration is easy, but the disadvantage is that only one node may initiate a transaction at a time even though there may be empty slots on the ring passing by another nodes, resulting in wasted bandwidth. *Register-insertion rings* were chosen for the IEEE SCI standard. Here, a bypass FIFO between the input and output stages of the ring is used to buffer incoming transactions (with backward flow-control to avoid overloading), while the local node is transmitting. When the local node finishes, the contents of bypass FIFO are forwarded on the output link, and the local node is not allowed to transmit until the bypass FIFO is empty. Multiple nodes may be transmitting at a time, and parts of the ring will stall when they are overloaded. Finally, in *slotted rings*, the ring is divided into transaction slots with labelled types (for different sized transactions such as requests and data replies), and these slots keep circulating around the ring. A processor ready to transmit a transaction waits until an empty slot of the required type comes by (indicated by a bit in the slot header), and then it inserts its message. A "slot" here really means a sequence of empty time slots, the length of the sequence depending on the type of message. In theory, the slotted ring restricts the utilization of the ring bandwidth by hard-wiring the mixture of available slots of different types, which may not match the actual traffic pattern for a given workload. However, for a given coherence protocol the mix of message types is reasonably well known and little bandwidth is wasted in practice [BaD93, BaD95].

While it may seem at first that broadcast and snooping wastes bandwidth on an interconnect such as a ring, in reality it is not so. A broadcast takes only twice as the average point-to-point message on a ring, since the latter between two randomly chosen nodes will traverse half the ring on

average. Also, broadcast is needed only for request messages (read-miss, write-miss, upgrade requests) which are all short; data reply messages are put on the ring by the source of the data and stop at the requesting node.

Consider a read miss in the cache. If the home of the block is not local, the read request is placed on the ring. If the home is local, we must determine if the block is dirty in some other node, in which case the local memory should not respond and a request should be placed on the ring. A simple solution is to place all misses on the ring, as on a bus-based design with physically distributed memory, such as the Sun Enterprise. Alternatively, a *dirty bit*, can be maintained for each block in home memory. This bit is turned ON if a block is cached in dirty state in some node other than the home. If the bit is on, the request goes on the ring. The read request now circles the ring. It is snooped by all nodes, and either the home or the dirty node will respond (again, if the home were not local the home node uses the dirty bit to decide whether or not it should respond to the request). The request and response transactions are removed from the ring when they reach the requestor. Write miss and write upgrade transactions also appear on the ring as requests, if the local cache state warrants that an invalidation request be sent out. Other nodes snoop these requests and invalidate their blocks if necessary. The return of the request to the requesting node serves as an acknowledgment. When multiple nodes attempt to write to the same block concurrently, the winner is the one that reaches the current *owner* of the block (home node if block is clean, else the dirty node) first; the other nodes are implicitly/explicitly sent negative acknowledgments (NAKs), and they must retry.

From an implementation perspective, a key difficulty with snoopy protocols on rings is the real-time constraints imposed: The snooper on the ring-interface must examine and react to all passing messages without excessive delay or internal queuing. This can be difficult for register-insertion rings, since many short request messages may be adjacent to each other in the ring. With rings operating at high speeds, the requests can be too close together for the snooper to respond to in a fixed time. The problem is simplified in slotted rings, where careful choice and placement of short request messages and long data response messages (the data response messages are point-to-point and do not need snooping) can ensure that request-type messages are not too close together [BaD95]. For example, slots can be grouped together in frames, and each frame can be organized to have request slots followed by response slots. Nonetheless, as with busses ultimately bandwidth on rings is limited by snoop bandwidth rather than raw data transfer bandwidth.

Coherence issues are handled as follows in snoopy ring protocols: (a) Enough information about the state in other nodes to determine whether to place a transaction on the ring is obtained from the location of the home, and from the dirty bit in memory if the block is locally allocated; (b) other copies are found through broadcast; and (c) communication with them happens simultaneously with finding them, through the same broadcast and snooping mechanism. Consistency is a bit trickier, since there is the possibility that processors at different points on the ring will see a pair of transactions on the ring in different orders. Using invalidation protocols simplifies this problem because writes only cause read-exclusive transactions to be placed on the ring and all nodes but the home node will respond simply by invalidating their copy. The home node can determine when conflicting transactions are on the ring and take special action, but this does increase the number of transient states in the protocol substantially.

6.7.5 Scaling Data Bandwidth and Snoop Bandwidth

There are several alternative ways to increase the bandwidth of SMP designs that preserve much of the simplicity of bus-based approaches. We have seen that with split-transaction busses, the arbitration, the address phase, and the data phase are pipelined, so each of them can go on simultaneously. Scaling data bandwidth is the easier part, the real challenge is scaling the snoop bandwidth.

Let's consider first scaling data bandwidth. Cache blocks are large compared to the address that describes them. The most straightforward to increase the data bandwidth is simple to make the data bus wider. We see this, for example, in the Sun Enterprise design which uses a 128-bit wide data bus. With this approach, a 32-byte block is transferred in only two cycles. The down side of this approach is cost; as the bus get wider it uses a larger connector, occupies more space on the board, and draws more power. It certainly pushes the limit of this style of design, since it means that a snoop operations, which needs to be observed by all the caches and acknowledged, must complete in only two cycles. A more radical alternative is to replace the data bus with a cross-bar, directly connecting each processor-memory module to every other one. It is only the address portion of the transaction that needs to be broadcast to all the nodes in order to determine the coherence operation and the data source, i.e., memory or cache. This approach is followed in the IBM PowerPC based RS6000 G30 multiprocessor. A bus is used for addresses and snoop results, but a cross-bar is used to move the actual data. The individual paths in the cross-bar need not be extremely wide, since multiple transfers can occur simultaneously.

A brute force way to scale bandwidth in a bus based system is simply to use multiple busses. In fact, this approach offers a fundamental contribution. In order to scale the snoop bandwidth beyond one coherence result per address cycle, there must be multiple simultaneous snoop operations. Once there are multiple address busses, the data bus issues can be handled by multiple data busses, cross-bars, or whatever. Coherence is easy. Different portions of the address space use different busses, typically each bus will serve specific memory banks, so a given address always uses the same bus. However, multiple address busses would seem to violate the critical mechanism used to ensure memory consistency – serialized arbitration for the address bus. Remember, however, that sequential consistency requires that there be a logical total order, not that the address events be in strict chronological order. A static ordering is assigned logically assigned to the sequence of busses. An address operation i logically preceeds j if it occurs before j in time or if they happen on the same cycle but i takes place on a lower numbered bus. This multiple bus approach is used in Sun SparcCenter 2000, which provided two split-phase (or packet switched) XDB busses, each identical to that used in the SparcStation 1000, and scales to 30 processors. The Cray CS6400 used four such busses and scales to 64 processors. Each cache controller snoops all of the busses and responds according to the cache coherence protocol. The Sun Enterprise 10000 combines the use of multiple address busses and data cross bars to scale to 64 processors. Each board consists of four 250 MHz processors, four banks of memory (up to 1 GB each), and two independent SBUS I/O busses. Sixteeen of these boards are connected by a 16x16 cross bar with paths 144 bits wide, as well as four address busses associated with the four banks on each board. Collectively this provide 12.6 GB/s of data bandwidth and a snoop rate of 250 MHz.

6.8 Concluding Remarks

The design issues that we have explored in this chapter are fundamental, and will remain important with progress in technology. This is not to say that the optimal design choices will not change. For example, while shared-cache architectures are not currently very popular, it is possible that sharing caches at some level of the hierarchy may become quite attractive when multichip-module packaging technology becomes cheap or when multiple processors appear on a single chip, as long as destructive interference does not dominate in the workloads of interest.

A shared bus interconnect clearly has bandwidth limitations as the number of processors or the processor speed increases. Architects will surely continue to find innovative ways to squeeze more data bandwidth and more snoop bandwidth out of these designs, and will continue to exploit the simplicity of a broadcast-based approach. However, the general solution in building scalable cache-coherent machines is to distribute memory physically among nodes and use a scalable interconnect, together with coherence protocols that do not rely on snooping. This direction is the subject of the subsequent chapters. It is likely to find its way down to even the small scale as processors become faster relative to bus and snoop bandwidth. It is difficult to predict what the future holds for busses and the scale at which they will be used, although they are likely to have an important role for some time to come. Regardless of that evolution, the issues discussed here in the context of busses—placement of the interconnect within the memory hierarchy, the cache coherence problem and the various coherence protocols at state transition level, and the correctness and implementation issues that arise when dealing with many concurrent transactions—are all largely independent of technology and are crucial to the design of all cache-coherent shared-memory architectures regardless of the interconnect used. Moreover, these designs provide the basic building block for larger scale design presented in the remainder of the book.

6.9 References

[AdG96]   Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, vol. 29, no. 12, December 1996, pp. 66-76.

[AgG88]   Anant Agarwal and Anoop Gupta. Memory-reference Characteristics of Multiprocessor Applications Under MACH. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, May 1988.

[ArB86]   James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[BaW88]   Jean-Loup Baer and Wen-Hann Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on   Computer Architecture*, pp. 73--80, May 1988.

[BJS88]   F. Baskett, T. Jermoluk, and D. Solomon, The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. *Proc of the 33rd IEEE Computer Society Int. Conf. - COMPCOM 88*, pp. 468-71, Feb. 1988.

[BRG+89]   David Black, Richard Rashid, David Golub, Charles Hill, Robert Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[CAH+93] Ken Chan, et al. Multiprocessor Features of the HP Corporate Business Servers. In *Proceedings of COMPCON*, pp. 330-337, Spring 1993.

[CoF93] Alan Cox and Robert Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 98-108, May 1993.

[DSB86] Michel Dubois, Christoph Scheurich and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 434-442.

[DSR+93] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 88-97, May 1993.

[DuL92] C. Dubnicki and T. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 170-180, May 1992.

[EgK88] Susan Eggers and Randy Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on   Computer Architecture*, pp. 373-382, May 1988.

[EgK89a] Susan Eggers and Randy Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270, May 1989.

[EgK89b] Susan Eggers and Randy Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 2-15, May 1989.

[FCS+93] Jean-Marc Frailong, et al. The Next Generation SPARC Multiprocessing System Architecture. In *Proceedings of COMPCON*, pp. 475-480, Spring 1993.

[GaW93] Mike Galles and Eric Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In Proceedings of 27th Annual Hawaii International Conference on Systems Sciences, January 1993.

[Goo83] James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on   Computer Architecture*, pp. 124-131, June 1983.

[Goo87] James Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987.

[GSD95] H. Grahn, P. Stenstrom, and M. Dubois, Implementation and Evaluation of Update-based Protocols under Relaxed Memory Consistency Models. In Future Generation Computer Systems, 11(3): 247-271, June 1995.

[HeP90] John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 1990.

[Her91] Maurice Herlihy. Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems, vol. 13, no. 1, January 1991, pp. 124-149.

[HiS87] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. IEEE Transactions on Computers, vol. C-38, no. 12, December 1989, pp. 1612-1630.

[HEL+86] Mark Hill et al. Design Decisions in SPUR. *IEEE Computer*, 19(10):8-22, November 1986.

[JeE91] Tor E. Jeremiassen and Susan J. Eggers. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 377-381.

[KMR+86] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator. Competitive Snoopy Caching. In Pro-

ceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986.

[Kro81]   D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pp. 81-87, May 1981.

[Lam79]   Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, vol. C-28, no. 9, September 1979, pp. 690-691.

[Lau94]   James Laudon. Architectural and Implementation Tradeoffs for Multiple-Context Processors. Ph.D. Thesis, Computer Systems Laboratory, Stanford University, 1994.

[Lei92]   C. Leiserson, et al. The Network Architecture of the Connection Machine CM-5. Symposium of Parallel Algorithms and Architectures, 1992.

[LLJ+92]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. IEEE Transactions on Parallel and Distributed Systems, 4(1):41-61, January 1993.

[McC84]   McCreight, E. The Dragon Computer System: An Early Overview. Technical Report, Xerox Corporation, September 1984.

[Mip91]   MIPS R4000 User's Manual. MIPS Computer Systems Inc. 1991.

[PaP84]   Mark Papamarcos and Janak Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348-354, June 1984.

[Ros89]   Bryan Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles*, December 1989.

[ScD87]   Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 234-243.

[SFG+93]   Pradeep Sindhu, et al. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. In *Proceedings of COMPCON*, pp. 338-344, Spring 1993.

[SHG93]   Jaswinder Pal Singh, John L. Hennessy and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, vol. 26, no. 7, July 1993.

[Smi82]   Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, September 1982.

[SwS86]   Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 414-423, May 1986.

[TaW97]   Andrew S. Tanenbaum and Albert S. Woodhull, Operating System Design and Implementation (Second Edition), Prentice Hall, 1997.

[Tel90]   Patricia Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26-36, June 1990.

[TBJ+88]   M. Thompson, J. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Proceedings of USENIX Technical Conference,* February 1988.

[TLS88]   Charles Thacker, Lawrence Stewart, and Edwin Satterthwaite, Jr. Firefly: A Multiprocessor Workstation, IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 909-20.

[TLH94]   Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651-663, June 1994.

[WBL89]   Wen-Hann Wang, Jean-Loup Baer and Henry M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In Proceedings of the 16th Annual International Symposium

on Computer Architecture, pp. 140-148, June 1989.

[WeS94] Shlomo Weiss and James Smith. Power and PowerPC. Morgan Kaufmann Publishers Inc. 1994.

[WEG+86] David Wood, et al. An In-cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 358-365, May 1986.

6.10 Exercises

6.1 shared cache versus private caches. Consider two machines M1 and M2. M1 is a four-proces-
sor shared-cache machine, while M2 is a four-processor bus-based snoopy cache machine.
M1 has a single shared 1 Mbyte two-way set-associative cache with 64-byte blocks, while
each processor in M2 has a 256 Kbyte direct-mapped cache with 64-byte blocks. M2 uses the
Illinois MESI coherence protocol. Consider the following piece of code:

```
double A[1024,1024]; /* row-major; 8-byte elems */
double C[4096];
double B[1024,1024];

for (i=0; i<1024; i+=1) /* loop-1 */
     for (j=myPID; j<1024; j+=numPEs)
          {
           B[i,j] = (A[i+i,j] + A[i-1,j] +
                          A[i,j+1] + A[i,j-1]) / 4.0;
          }
for (i=myPID; i<1024; i+=numPEs) /* loop-2 */
     for (j=0; j<1024; j+=1)
          {
           A[i,j] = (B[i+i,j] + B[i-1,j] +
                          B[i,j+1] + B[i,j-1]) / 4.0;
          }
```

a. Assume that the array A starts at address 0x0, array C at 0x300000, and array B at
0x308000. Assume that all caches are initially empty. Assume each processor executes the
above code, and that myPID varies from 0-3 for the four processors. Compute misses for
M1, separately for loop-1 and loop-2. Do the same for M2, stating any assumptions that
you make.

b. Briefly comment on how your answer to part **a.** would change if the array C were not
present. State any other assumptions that you make.

c. What can be learned about advantages and disadvantages of shared-cache architecture
from this exercise?

d. Given your knowledge about the Barnes-Hut, Ocean, Raytrace, and Multiprog workloads
from previous chapters and data in Section 5.5, comment on how each of the applications
would do on a four-processor shared-cache machine with a 4 Mbyte cache versus a four
processor snoopy-bus-based machine with1 Mbyte caches. It might be useful to verify
your intuition using simulation.

e. Compared to a shared first-level cache, what are the advantages and disadvantages of hav-
ing private first-level caches, but a shared second-level cache? Comment on how modern
microprocessors, for example, MIPS R10000 and IBM/Motorola PowerPC 620, encour-
age or discourage this trend. What would be the impact of packaging technology on such
designs?

6.2 Cache inclusion.

a. Using terminology in Section 6.4, assume both L1 and L2 are 2-way, and $n2 > n1$, and
$b1=b2$, and replacement policy is FIFO instead of LRU. Does inclusion hold? What if it is
random, or based on a ring counter.

b. Give an example reference stream showing inclusion violation for the following situations:

(i) L1 cache is 32 bytes, 2-way set-associative, 8-byte cache blocks, and LRU replacement. L2 cache is 128 bytes, 4-way set-associative, 8-byte cache blocks, and LRU replacement.

(ii) L1 cache is 32 bytes, 2-way set-associative, 8-byte cache blocks, and LRU replacement. L2 cache is 128 bytes, 2-way set-associative, 16-byte cache blocks, and LRU replacement.

c. For the following systems, state whether or not the caches provide for inclusion: If not, state the problem or give an example that violates inclusion.

(i) Level 1: 8KB Direct mapped primary instruction cache, 32 byte line size
       8KB Direct mapped primary data cache, write through, 32 byte line size
Level 2:   4MB 4 way set associative unified secondary cache, 32 byte line size

(ii) Level 1: 16KB Direct Mapped unified primary cache, write-through, 32 byte line size
Level 2:   4MB 4 way set associative unified secondary cache, 64 byte line size

d. The discussion of the inclusion property in Section 6.4 stated that in a common case inclusion is satisfied quite naturally. The case is when the L1 cache is direct-mapped ($a_1 = 1$), L2 can be direct-mapped or set associative ($a_2 >= 1$) with any replacement policy (e.g., LRU, FIFO, random) as long as the new block brought in is put in both L1 and L2 caches, the block-size is the same ($b_1 = b_2$), and the number of sets in the L1 cache is equal to or smaller than in L2 cache ($n_1 =< n_2$). Show or argue why this is true.

6.3 Cache tag contention <> Assume that each processor has separate instruction and data caches, and that there are no instruction misses. Further assume that, when active, the processor issues a cache request every 3 clock cycles, the miss rate is 1%, miss latency is 30 cycles. Assume that tag reads take one clock cycle, but modifications to the tag take two clock cycles.

a. Quantify the performance lost to tag contention if a single-level data cache with only one set of cache tags is used. Assume that the bus transactions requiring snoop occur every 5 clock cycles, and that 10% of these invalidate a block in the cache. Further assume that snoops are given preference over processor accesses to tags. Do back-of-the-envelope calculations first, and then check the accuracy of your answer by building a queuing model or writing a simple simulator.

b. What is the performance lost to tag contention if separate sets of tags for processor and snooping are used?

c. In general, would you decide to give priority in accessing tags to processor references or bus snoops?

6.4 Protocol and Memory System Implementation: SGI Challenge.

a. The designers of the SGI Challenge multiprocessor considered the following bus controller optimization to make better use of interleaved memory and bus bandwidth. If the controller finds that a request is already outstanding for a given memory bank (which can be determined from the request table), it does not issue that request until the previous one for that bank is satisfied. Discuss potential problems with this optimization and what features in the Challenge design allow this optimization.

b. The Challenge multiprocessor's Powerpath-2 bus allows for eight outstanding transactions. How do you think the designers arrived at that decision? In general, how would you

determine how many outstanding transactions should be allowed by a split-transaction bus. State all your assumptions clearly.

c. Although the Challenge supports the MESI protocol states, it does not support the cache-to-cache transfer feature of the original Illinois MESI protocol.

(i) Discuss the possible reasons for this choice.

(ii) Extend the Challenge implementation to support cache-to-cache transfers. Discuss extra signals needed on bus, if any, and keep in mind the issue of fairness.

d. Although the Challenge MESI protocol has four states, the tags stored with the cache controller chip keep track of only three states (I, S, and E+M). Explain why this is still works correctly. Why do you think that they made this optimization?

e. The main memory on the Challenge speculatively initiates fetching the data for a read request, even before it is determined if it is dirty in some processor's cache. Using data in Table 5-3 estimate the fraction of useless main memory accesses. Based on the data, are you in favor of the optimization? Are these data methodologically adequate? Explain.

f. The bus interfaces on the SGI Challenge support request merging. Thus, if multiple processors are stalled waiting for the same memory block, then when the data appears on the bus, all of them can grab that data off the bus. This feature is particularly useful for implementing spin-lock based synchronization primitives. For a test-test&set lock, show the minimum traffic on the bus with and without this optimization. Assume that there are four processors, each acquiring lock once and then doing an unlock, and that initially no processor had the memory block containing the lock variable in its cache.

g. Discuss the cost, performance, implementation, and scalability tradeoffs between the multiple bus architecture of the SparcCenter versus the single fast-wide bus architecture of the SGI Challenge, as well as any implications for program semantics and deadlock.

6.5 Split Transaction Busses.

a. The SGI Challenge bus allows for eight outstanding transactions. How did the designers arrive at that decision? To answer that, suggest a general formula to indicate how many outstanding transactions should be supported given the parameters of the bus. Use the following parameters:

P   Number of processors
M   Number of memory banks
L   Average memory latency (cycles)
B   Cache block size (bytes)
W   Data bus width (bytes)

Define any other parameters you think are essential. *Keep your formula simple*, clearly state any assumptions, and justify your decisions.

a. To improve performance of the alternative design for supporting coherence on a split-transaction bus (discussed at the end of Section 6.5), a designer lays down the following design objectives: (i) the snoop results are generated in order, (ii) the data responses, however, may be out of order, and (iii) there *can* be multiple pending requests to the same memory block. Discuss in detail the implementation issues for such an option, and how it compares to the base implementation discussed in Section 6.5.

b. In the split-transaction solution we have discussed in Section 6.5, depending on the processor-to-cache interface, it is possible that an invalidation request comes immediately after the data response, so that the block is invalidated before the processor has had a chance to actually access that cache block and satisfy its request. Why might this be a problem and how can you solve it?

c. When supporting lock-up-free caches, a designer suggests that we also add more entries to the request-table sitting on the bus-interface of the split-transaction bus. Is this a good idea and do you expect the benefits to be large?

d. <<The ways to preserve SC with multiple outstanding transactions and commit versus complete. Apply them to Example 6-3 on page 381 and convince yourself that they work. Under what conditions is one solution better than the other? Answer: For SC, preserving order may be better without the optimizations since need to check/flush frequently in second case. Particularly with multi-level caches.

6.6 Computing bandwidth needs using data provided. Assume a system bus similar to Powerpath2, as discussed in Section 6.6. Assuming 200-MIPS/200-MFLOPS processors with 1 Mbyte caches and 64-byte cache blocks, for each of the applications in Table 5-1 compute the bus bandwidth when using:

a. The Illinois MESI protocol.

b. The Dragon protocol.

c. The Illinois MESI protocol assuming 256-byte cache blocks.

For each of the above parts, compute the utilization of the address+command bus separately from the utilization of the data bus. State all assumptions clearly.

d. Do parts a, and b for a single SparcCenter XDBus, which has 64-bit wide multiplexed address and data signals. Assume that the bus runs at 100MHz, and that transmitting address information takes 2 cycles on the bus, and that 64bytes of data takes 9 cycles on the bus.

6.7 Multi-level Caches

a. One deadlock solution proposed for multi-level caches in Section 6.5.8 is to make all queues 9 deep. Can the queues be smaller? If so, why? Discuss, why it may be beneficial to have deeper queues than the size required by deadlock considerations.

b. [Section 6.4 presents coherence protocols assuming 2-level caches. What if there are three or more levels in the cache hierarchy. Extend the Illinois MESI protocol for the middle cache in a 3-level hierarchy. List any additional states or actions needed, and present the state-transition diagram. Discuss the implementation details, including the number and nature of intervening queues.

6.8 TLB Shootdown. Figure 6-26 shows the details of the TLB shootdown algorithm used in the Mach operating system [BRG+89]. The basic data structures are as follows. For each processor, the following data structures are maintained: (i) an *active* flag indicating whether the processor is actively using any page tables; (ii) a queue of TLB flush notices indicating the range of virtual addresses whose mappings are to be changed; and (iii) a list indicating currently active page tables, i.e., processes whole PTEs may be cached in the TLB. For every page table, there is: (i) a spinlock that processor must hold while making changes to that page table, and (ii) a set of processors on which this page table is currently active. While the basic shootdown approach is simple, practical implementations require careful sequencing of steps and locking of data structures.
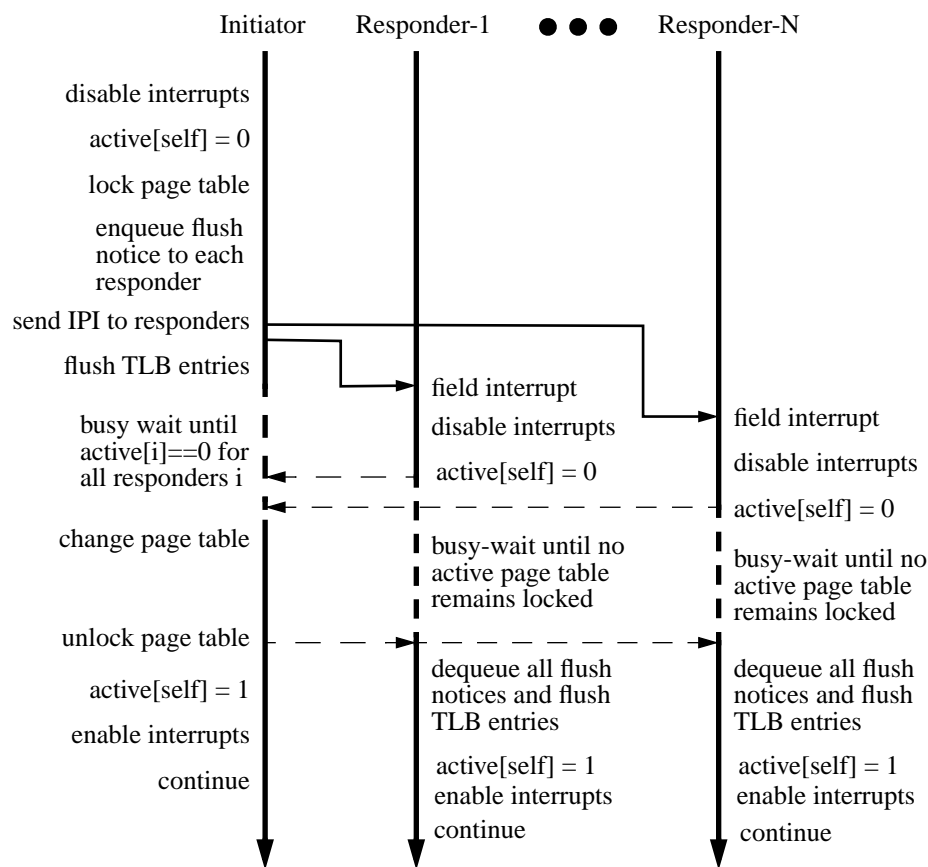
Figure 6-26 The Mach TLB shootdown algorithm.

The initiator is the processor making changes to the page-table, while the responders are all other processors that may have entries from that page-table cached.

a. Why are page table entries modified before sending interrupts or invalidate messages to other processors in TLB coherence?

b. Why must the initiator of the shootdown in Figure 6-26 mask out inter-processor-interrupts (IPIs) before acquiring the page table lock, and to clear its own active flag before acquiring the page table lock? Can you think of any deadlock conditions that exist in the figure, and if so how would you solve them?

c. A problem with the Mach algorithm is that it makes all responders busy-wait while the initiator makes changes to the page table. The reason is that it was designed for use with microprocessors that autonomously wrote back the entire TLB entry into the corresponding PTE whenever the usage/dirty bit was set. Thus, for example, if other processors were allowed to use the page table while the initiator was modifying it, an autonomous writeback from those processors could overwrite the new changes. How would you design the TLB hardware and/or algorithm so that responders do not have to busy-wait? [One solution was used in the IBM RP3 system [Ros89].

d. For MACH shootdown we say it would be better to update the use and dirty information for pages in software. (Machines based on the MIPS processor architecture actually do all of this in software.) Lookup the operating system of a MIPS-based machine or suggest how you would write the TLB fault handlers so that the usage and dirty information was made available to the OS page replacement algorithms and for writeback of dirty pages.

e. Under what circumstances would it be better to flush the whole TLB versus selectively trying to invalidate TLB entries?

CHAPTER 7   Scalable Multiprocessors

7.1 Introduction

In this chapter we begin our study of the design of machines that can be scaled in a practical manner to hundreds or even thousands of processors. Scalability has profound implications at all levels of the design. For starters, it must be physically possible and technically feasible to construct a large configuration. Adding processors clearly increases the potential computational capacity of the system, but to realize this potential, the memory bandwidth must scale with the number of processors. A natural solution is to distribute the memory with the processors as in our generic multiprocessor, so each processor has direct access to local memory. However, the communication network connecting these nodes must provide scalable bandwidth at reasonable latency. In addition, the protocols used in transferring data within the system must scale, and so must the techniques used for synchronization. With scalable protocols on a scalable network, a very large number of transactions can take place in the system simultaneously and we cannot rely on global information to establish ordering or to arbitrate for resources. Thus, to achieve scalable application performance, scalability must be addressed as a "vertical problem" throughout each of the layers of the system design.

Let's consider a couple of familiar design points to make these scalability issues more concrete. The small-scale shared memory machines described in the previous chapter can be viewed as one extreme point. A shared bus typically has a maximum length of a foot or two, a fixed number of slots, and a fixed maximum bandwidth, so it is fundamentally limited in scale. The interface to the communication medium is an extension of the memory interface, with additional control lines and controller states to effect the consistency protocol. A global order is established by arbitra-

tion for the bus and a limited number of transactions can be outstanding at any time. Protection is enforced on all communication operations through the standard virtual-to-physical address translation mechanism. There is total trust between processors in the system, which are viewed as under the control of a single operating system that runs on all of the processors, with common system data structures. The communication medium is contained within the physical structure of the box and is thereby completely secure. Typically, if any processor fails the system is rebooted. There is little or no software intervention between the programming model and the hardware primitives. Thus, at each level of the system design decisions are grounded in scaling limitations at layers below and assumptions of close coupling between the components. Scalable machines are fundamentally less closely coupled than the bus-based shared-memory multiprocessors of the previous chapter and we are forced to rethink how processors interact with other processors and with memories.

At the opposite extreme we might consider conventional workstations on a local-area or even wide-area network. Here there is no clear limit to physical scaling and very little trust between processors in the system. The interface to the communication medium is typically a standard peripheral interface at the hardware level, with the operating system interposed between the user-level primitives and the network to enforce protection and control access. Each processing element is controlled by its own operating system, which treats the others with suspicion. There is no global order of operations and consensus is difficult to achieve. The communication medium is external to the individual nodes and potentially insecure. Individual workstations can fail and restart independently, except perhaps where one is providing services to another. There is typically a substantial layer of software between the hardware primitives and any user-level communication operations, regardless of programming model, so communication latency tends to be quite high and communication bandwidth low. Since communication operations are handled in software, there is no clear limit on the number of outstanding transactions or even the meaning of the transactions. At each level of the system design it is assumed that communication with other nodes is slow and inherently unreliable. Thus, even when multiple processors are working together on a single problem, it is difficult to exploit the inherent coupling and trust within the application to obtain greater performance from the system.

Between these extremes there is a spectrum of reasonable and interesting design alternatives, several of which are illustrated by current commercial large-scale parallel computers and emerging parallel computing clusters. Many of the *massively parallel processing systems* (MPPs) employ sophisticated packaging and a fast dedicated proprietary network, so that a very large number of processors can be located in a confined space with a high-bandwidth and low-latency communication. Other scalable machines use essentially conventional computers as nodes, with more or less standard interfaces to fast networks. In either case, there is a great deal of physical security and the option of either a high degree of trust or of substantial autonomy.

The generic multiprocessor of Chapter 1 provides a useful framework for understanding scalable designs: the machine is organized as essentially complete computational nodes, each with a memory subsystem and one or more processors, connected by a scalable network. One of the most critical design issues is the nature of the *node-to-network interface*. There is a wide scope of possibilities, differing in how tightly coupled the processor and memory subsystems are to the network and in the processing power within the network interface itself. These issues affect the degree of trust between the nodes and the performance characteristics of the communication primitives, which in turn determine the efficiency with which various programming models can be realized on the machine.

Our goal in this chapter is to understand the design trade-offs across the spectrum of communication architectures for large scale machines. We want to understand, for example, how the decision to pursue a more specialized or a more commodity approach impacts the capabilities of the node-to-network interface, the ability to support various programming models efficiently, and the limits to scale. We begin in Section 7.2 with a general discussion of scalability, examining the requirements it places on system design in terms of bandwidth, latency, cost, and physical construction. This discussion provides a nut-and-bolts introduction to a range of recent large scale machines, but also allows us to develop an abstract view of the scalable network on the "other side" of the node-to-network interface. We return to an in-depth study of the design of scalable networks later in Chapter 7, after having fully developed the requirements on the network in this and the following chapter.

We focus in Section 7.3 on the question of how programming models are realized in terms of the communication primitives provided on large-scale parallel machines. The key concept is that of a *network transaction*, which is the analog for scalable networks of the bus transaction studied in the previous chapter. Working with a fairly abstract concept of a network transaction, we show how shared address space and message passing models are realized through *protocols* built out of network transactions.

The remainder of the chapter examines a series of important design points with increasing levels of direct hardware interpretation of the information in the network transaction. In a general sense, the interpretation of the network transaction is akin to the interpretation of an instruction set. Very modest interpretation suffices in principle, but more extensive interpretation is important for performance in practice. Section 7.4 investigates the case where there is essentially no interpretation of the message transaction; it is viewed as a sequence of bits and transferred blindly into memory via a physical DMA operation under operating system control. This is an important design point, as it represents many early MPPs and most current local-area network (LAN) interfaces.

Section 7.5 considers more aggressive designs where messages can be sent from user-level to user-level without operating system intervention. At the very least, this requires that the network transaction carry a user/system identifier, which is generated by the source communication assist and interpreted by the destination. This small change gives rise to the concept of a user virtual network, which, like virtual memory, must present a protection model and offer a framework for sharing the underlying physical resources. A particularly critical issue is user-level message reception, since message arrival is inherently asynchronous to the user thread for which it is destined.

Section 7.6 focuses on designs that provide a global virtual address space. This requires substantial interpretation at the destination, since it needs to perform the virtual-to-physical translation, carry out the desired data transfer and provide notification. Typically, these designs use a dedicated message processor, so extensive interpretation of the network transaction can be performed without the specifics of the interpretation being bound at machine design-time. Section 7.7 considers more specialized support for a global physical address space. In this case, the communication assist is closely integrated with the memory subsystem and, typically, it is a specialized device supporting a limited set of network transactions. The support of a global physical address space brings us full circle to designs that are close in spirit to the small-scale shared memory machines studied in the previous chapter. However, automatic replication of shared data through coherent caches in a scalable fashion is considerably more involved than in the bus-based setting, and we devote all of Chapter 6 to that topic.

7.2 Scalability

What does it mean for a design to "scale". Almost all computers allow the capability of the system to be increased in some form, for example by adding memory, I/O cards, disks, or upgraded processor(s), but the increase typically has hard limits. A scalable system attempts to avoid inherent design limits on the extent to which resources can be added to the system. In practice a system can be quite scalable even if it is not possible to assemble an arbitrarily large configuration because at any point in time, there are crude limits imposed by economics. If a "sufficiently large" configuration can be built, the scalability question has really to do with the incremental cost of increasing the capacity of the system and the resultant increase in performance delivered on applications. In practice no design scales perfectly, so our goal is to understand how to design systems that scale up to a large number of processors effectively. In particular, we look at four aspects of scalability in a more or less top-down order. First, how does the bandwidth or throughput of the system increase with additional processors? Ideally, throughput should be proportional to the number of processors. Second, how does the latency or time per operation increase? Ideally, this should be constant. Third, how does the cost of the system increase, and finally, how do we actually package the systems and put them together?

It is easy to see that the bus-based multiprocessors of the previous chapter fail to scale well in all four aspects, and the reasons are quite interrelated. In those designs, a number of processors and memory modules were connected via a single set of wires, the bus. When one module is driving a wire, no other module can drive it. Thus, the bandwidth of a bus does not increase as more processors are added to it; at some point it will saturate. Even accepting this defect, we could consider constructing machines with many processors on a bus, perhaps under the belief that the bandwidth requirements per processor might decrease with added processor. Unfortunately, the clock period of the bus is determined by the time to drive a value onto the wires and have it sampled by every module on the bus, which increases with the number of modules on the bus and with wire length. Thus, a bigger bus would have longer latency and *less* aggregate bandwidth. In fact, the signal quality on the wire degrades with length and number of connectors, so for any bus technology there is a hard limit on the number of slots into which modules can be plugged and the maximum wire length. Accepting this limit, it would seem that the bus-based designs have good cost scaling, since processors and memory can be added at the cost of the new modules. Unfortunately, this simple analysis overlooks that even the minimum configuration is burdened by a large fix cost for the infrastructure needed to support the maximum configuration; the bus, the cabinet, the power supplies, and other components must be sized for the full configuration. At the very least, a scalable design must overcome these limitations – the aggregate bandwidth must increase with the number of processors, the time to perform an operation should not increase substantially with the size of the machine, and it must be practical and cost-effective to build a large configuration. In addition, it is good if the design scales down well, also.

7.2.1 Bandwidth Scaling

Fundamentally, if a large number of processors are to exchange information simultaneously with many other processors or memories, there must be a large number of *independent* wires connecting them. Thus, scalable machines must be organized in the manner illustrated abstractly by Figure 7-2; a large number of processor modules and memory modules connected together by independent wires (or links) through a large number of *switches*. We use the term switch in a very general sense to mean a device connecting a limited number of inputs to a limited number of out-

puts. Internally, such a switch may be realized by a bus, a cross-bar, or even an *ad hoc* collection of multiplexors. We call the number of outputs (or inputs) the *degree* of the switch. With a bus, the physical and electrical constraints discussed above determine its degree. Only a fraction of the inputs can transfer information to the outputs at once. A cross-bar allows every input to be connected to a distinct output, but the degree is constrained by the cost and complexity of the internal array of cross-points. The cost of multiplexors increases rapidly with the number of ports and latency increases as well. Thus, switches are limited in scale, but may be interconnected to form large configurations, i.e., *networks*. In addition to the physical interconnect between inputs and outputs, there must also be some form of controller to determine which inputs are to be connected to which outputs at each instant in time. In essence, a scalable network is like a roadway system with wires for streets, switches for intersections, and a simple way of determining which cars proceed at each intersection. If done right, a large number of vehicles may make progress to their destinations simultaneously and can get there quickly.
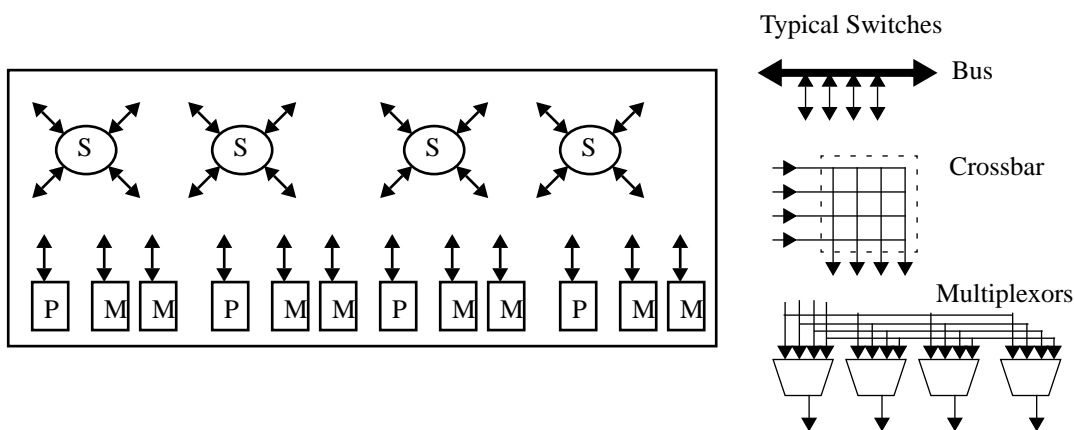
Figure 7-1 Abstract view of a scalable machine.

A large number of processor (P) and memory (M) modules connected together by independent wires (or links) through a large number of switch modules (S), each with some limited number of degree. An individual switch may be formed by a bus, a crossbar, multiplexors, or some other controlled connection between inputs and outputs.

By our definition, a basic bus-based SMP contains a single switch connecting the processors and the memories, and a simple hierarchy of bus-based switches connect these components to the peripherals. (We also saw a couple of machines in Chapter 4 in which the memory "bus" is a shallow hierarchy of switches.) The control is rather specialized in that the address associated with a transaction at one of the inputs is broadcast to all of the outputs and the acknowledgment determines which output is to participate. A *network switch* is a more general purpose device, in which the information presented at the input is enough for the switch controller to determine the proper output without consulting all the nodes. Pairs of modules are connected by *routes* through network switches.

The most common structure for scalable machines is illustrated by our generic architecture of Figure 7-2, in which one or more processors are packaged together with one or more memory modules and a communication assist as an easily replicated unit, which we will call a *node*. The "intranode" switch is typically a high-performance bus. Alternatively, systems may be constructed in a "dance hall" configuration, in which processing nodes are separated from memory nodes by the network, as in Figure 7-3. In either case, there is a vast variety of potential switch

designs, interconnection network topologies, and routing algorithms, which we will study in Chapter 10. The key property of a scalable network is that it provide a large number of independent communication paths between nodes such that the bandwidth increases as nodes are added. Ideally, the latency in transmitting data from one node to another should not increase with the number of nodes, and neither would the cost per node, but, as we discuss below, some increase in latency and cost is unavoidable.
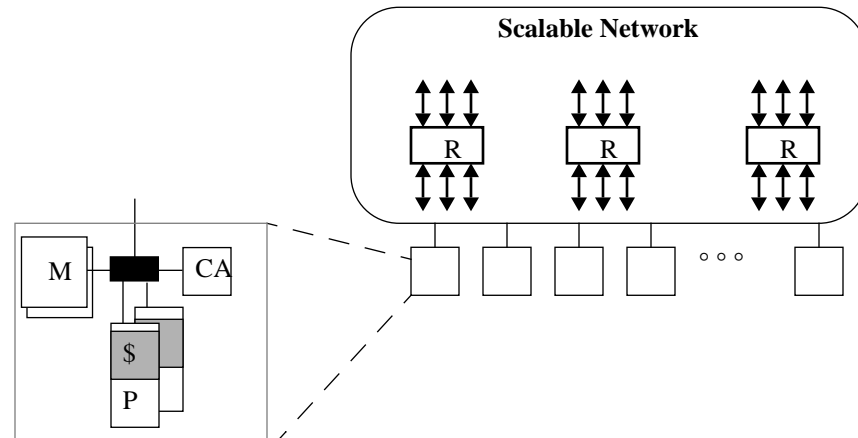
Figure 7-2 Generic distributed memory multiprocessor organization.

A collection of essentially complete computers, including processor and memory, communicating through a general purpose, high-performance, scalable interconnection network. Typically, each node contains a controller which assists in initiating and receiving communication operations.

If the memory modules are on the opposite side of the interconnect, as in Figure 7-3, even when there is no communication between processes the network bandwidth requirement scales linearly with the number of processors. Even with adequate bandwidth scaling, the computational performance may not scale perfectly because the access latency increases with the number of processors. By distributing the memories across the processors all processes can access local memory with fixed latency and bandwidth, independent of the number of processors, thus the computational performance of the system can scale perfectly, at least in this simple case. The network needs to meet the demands associated with actual communication and sharing of information. How computational performance scales in the more interesting case where processes do communicate depends on how the network itself scales, how efficient is the communication architecture, and how the program communicates.

To achieve scalable bandwidth we must abandon several key assumptions employed in bus-based designs, namely that there are a limited number of concurrent transactions, and that these are globally ordered via central arbitration and globally visible. Instead, it must be possible to have a very large number of concurrent transactions using different wires. They are initiated independently and without global arbitration. The effects of a transaction, such as changes of state, are directly visible only by the nodes involved in the transaction. (The effects may eventually become visible to other nodes as they are propagated by additional transactions.) Although it is possible to broadcast information to all the nodes, broadcast bandwidth, i.e., the rate at which
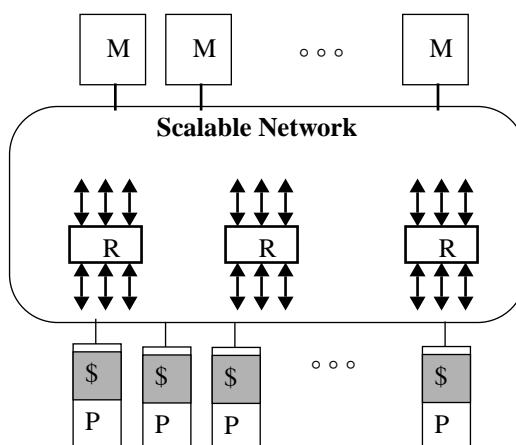
Figure 7-3 Dancehall multiprocessor organization.

Processors access memory modules across a scalable interconnection network. Even if processes are totally independent with no communication or sharing, the bandwidth requirement on the network increases linearly with the number of processors.

broadcasts can be performed, does not increase with the number of nodes. Thus, in a large system broadcasts can be used only infrequently.

7.2.2 Latency Scaling

We may extend our abstract model of scalable networks to capture the primary aspects of communication latency. In general, the time to transfer n bytes between two nodes is given by

$$T(n) = Overhead + Channel\ Time + Routing\ Delay,\qquad \textbf{(EQ 7.1)}$$

where Overhead is the processing time in initiating or completing the transfer, Channel Time is $\frac{n}{B}$, where B is the bandwidth of the "thinnest channel", and the routing delay is a function $f(H, n)$ of the number of routing steps, or "hops," in the transfer and possibly the number of bytes transferred.

The processing overhead may be fixed or it may increase with n, if the processor must copy data for the transfer. For most networks used in parallel machines, there is a fixed delay per router hop, independent of the transfer size, because the message "cuts through" several switches.[1] In contrast, traditional data communications networks *store-and-forward* the data at each stage, incurring a delay per hop proportional to the transfer size.[2] The store-and-forward approach is

---

1. The message may be viewed as a train, where the locomotive makes a choice at each switch in the track and all the cars behind follow, even though some may still be crossing a previous switch when the locomotive makes a new turn.

impractical in large-scale parallel machines. Since network switches have a fixed degree, the average routing distance between nodes must increase as the number of nodes increases. Thus, communication latency increases with scale. However, the increase may be small compared to the overhead and transfer time, if the switches are fast and the interconnection topology is reasonable. Consider the following example.

Example 7-1 Many classic networks are constructed out of fixed sized switches in a configuration, or *topology*, such that for n nodes, the distance from any network input to any network output is $\log n$ (base 2) and the total number of switches is $\alpha n \log n$, for some small constant α. Assuming the overhead is 1μs per message, the link bandwidth is 64 MB/s, and the router delay is 200ns per hop. How much does the time for a 128 byte transfer increase as the machine is scaled from 64 to 1,024 nodes?

Answer At 64 nodes, six hops are required so $T_{64}(128) = 1.0\mu s + \dfrac{128 \text{ B}}{64 \text{ B}/\mu s} + 6 \times 0.200\mu s = 4.2\mu s$. This increases to 5μs on a 1024 node configuration. Thus, the latency increases by less than 20% with a 16-fold increase in machine size. Even with this small transfer size, a store-and-forward delay would add 2 μs (the time to buffer 128 bytes) to the routing delay per hop.

Thus, the latency would be $T^{sf}_{64}(128) = 1.0\mu s + \left(\dfrac{128 \text{ B}}{64 \text{ B}/\mu s} + 0.200\mu s \right) \times 6 = 14.2\mu s$

at 64 nodes and $T^{sf}_{1024}(128) = 1.0\mu s + \left(\dfrac{128 \text{ B}}{64 \text{ B}/\mu s} + 0.200\mu s \right) \times 10 = 23\mu s$ at 1024 nodes.

In practice there is an important connection between bandwidth and latency that is overlooked in the preceding simplistic example. If two transfers involve the same node or utilize the same wires within the network, one may delay the other due to *contention* for the shared resource. As more of the available bandwidth is utilized, the probability of contention increases and the expected latency increases. In addition, queues may build up within the network, further increasing the expected latency. This basic saturation phenomenon occurs with any shared resource. One of the goals in designing a good network is to ensure that these load-related delays are not too large on the communication patterns that occur in practice. However, if a large number of processors transfer data to the same destination node at once, there is no escaping contention. The problem must be resolved at a higher level by balancing the communication load using the techniques described in Chapter 2.

---

2. The store-and-forward approach is like what the train does when it reaches the station. The entire train must come to a stop at a station before it can resume travel, presumably after exchanging goods or passengers. Thus, the time a given car spends in the station is linear in the length of the train.

7.2.3 Cost Scaling

For large machines, the scaling of the cost of the system is quite important. In general, we may view this as a fixed cost for the system infrastructure plus an incremental cost of adding processors and memory to the system, $Cost(p, m) = FixedCost + IncrementalCost(p, m)$. The fixed and incremental costs are both important. For example, in bus-based machines the fixed cost typically covers the cabinet, power supply, and bus supporting a full configuration. This puts small configurations at a disadvantage relative to the uniprocessor competition, but encourages expansion as the incremental cost of adding processors is constant and often much less than the cost of a stand-alone processor. (Interestingly, for most commercial bus-based machines the typical configuration is about one half of the maximum configuration. This is sufficiently large to amortize the fixed cost of the infrastructure, yet leaves "headroom" on the bus and allows for expansion. Vendors that supply "large" SMPs, say twenty or more processors, usually offer a smaller model providing a low-end sweetspot.) We have highlighted the incremental cost of memory in our cost equation because memory often accounts for a sizable fraction of the total system cost and a parallel machine need not necessarily have p times the memory of a uniprocessor.

Experience has shown that scalable machines must support a wide range of configurations, not just large and extra-large sizes. Thus, the "pay up-front" model of bus-based machines is impractical for scalable machines. Instead, the infrastructure must be modular, so that as more processors are added, more power supplies and more cabinets are added, as well as more network. For networks with good bandwidth scaling and good latency scaling, the cost of the network grows more than linearly with the number of nodes, but in practice the growth need not be much more linear.

Example 7-2   In many networks the number of network switches scales as $n \log n$ for n nodes. Assuming that at 64 nodes the cost of the system is equally balanced between processors, memory, and network, what fraction of the cost of a 1,024 node system is devoted to the network (assuming the same amount of memory per processor).

Answer   We may normalize the cost of the system to the per-processor cost of the 64-node system. The large configuration will have 10/6 as many routers per processor as the small system. Thus, assuming the cost of the network is proportional to the number of routers, the normalized cost per processor of the 1024-node system is

$$1\,\text{processor} \times 0.33 + 1\,\text{memory} \times 0.33 + \frac{10}{6}\text{routers} \times 0.33 \ = \ 1.22\,.$$ As the system is scaled up by 16-fold, the share of cost in the network increases from 33% to 45%. (In practice, there may be additional factors which cause network cost to increase somewhat faster than the number of switches, such as increased wire length.)

Network designs differ in how the bandwidth increases with the number of ports, how the cost increases, and how the delay through the network increases, but invariably, all three do increase. There are many subtle trade-offs among these three factors, but for the most part the greater the increase in bandwidth, the smaller the increase in latency, and the greater the increase in cost. Good design involves trade-offs and compromises. Ultimately, these will be rooted in the application requirements of the target workload.

Finally, when looking at issues of cost it is natural to ask whether a large-scale parallel machine can be cost effective, or is it only a means of achieving greater performance than is possible in smaller or uniprocessor configurations. The standard definition of efficiency ($Efficiency(p) = Speedup(p)/p$) reflects the view that a parallel machine is effective only if all its processors are effectively utilized all the time. This processor-centric view neglects to recognize that much of the cost of the system is elsewhere, especially in the memory system. If we define the cost scaling of a system, *costup*, in a manner analogous to speedup, ($Costup(p) = cost(p)/cost(1)$), then we can see that parallel computing is more cost-effective, i.e., has a smaller cost/performance, whenever $Speedup(p) > Costup(p)$. Thus, in a real application scenario we need to consider the entire cost of the system required to run the problem of interest.

7.2.4 Physical scaling

While it is generally agreed that modular construction is essential for large scale machines, there is little consensus on the specific requirements of physical scale, such as how compact the nodes need to be, how long the wires can be, the clocking strategy, and so on. There are commercial machines in which the individual nodes occupy scarcely more than the microprocessor footprint, while in others a node is a large fraction of a board or a complete workstation chassis. In some machines no wire is longer than a few inches, while in others the wires are several feet long. In some machines the links are one bit wide, in others eight or sixteen. Generally speaking, links tend to get slower with length and each specific link technology has an upper limit on length due to power requirements, signal-to-noise ratio, etc. Technologies that support very long distances, such as optical fiber, tend to have a much larger fixed cost per link for the transceivers and connectors. Thus, there are scaling advantages to a dense packing of nodes in physical space. On the other hand, a looser packing tends to reduce the engineering effort by allowing greater use of commodity components, which also reduces the time lag from availability of new microprocessor and memory technology to its availability in a large parallel machine. Thus, loose packing can have better technology scaling. The complex set of trade-offs between physical packaging strategies has given rise to a broad spectrum of designs, so it is best to look at some concrete examples. These examples also help make the other aspects of scaling more concrete.

Chip-level integration

A modest number of designs have integrated the communications architecture directly into the processor chip. The nCUBE/2 is a good representative of this densely packed node approach, even though the machine itself is rather old. The highly integrated node approach is also used in the MIT J-machine[Dal93] and number of other research machines and embedded systems. The design style may gain wider popularity as chip density continues to increase. In the nCUBE, each node had the processor, memory controller, network interface and a network router integrated in a single chip. The node chip connected directly to DRAM chips and fourteen bidirectional network links on a small card occupying a few square inches, shown in actual size in Figure 7-3. The network links form bit-serial channels connecting directly to other nodes[1] and one bidirectional channel to the I/O system. Each of the twenty-eight wires has a dedicated DMA device on the

---

1. The nCUBE nodes were connected in a in a hypercube configuration. A hypercube or n-cube is a graph generalizing the cube shown in Figure 7-12, where each node connects directly to $\log n$ other nodes in a n node configuration. Thus, 13 links could support a design up to 8096 nodes.

node chip. The nodes were socketed 64 to a board and the boards plug into a passive wiring back-plane, forming a direct node-to-node wire between each processor chip and $\log n$ other processors. The I/O links, one per processor, were brought outside the main rack to I/O nodes containing a node-chip (connecting to eight processors) and an I/O device. The maximum configuration was 8096 processors, and machines with 2048-nodes built in 1991. The system ran on a single 40 MHz clock in all configurations. Since some of the wires reached across the full width of the machine, dense packing was critical to limiting the maximum length.
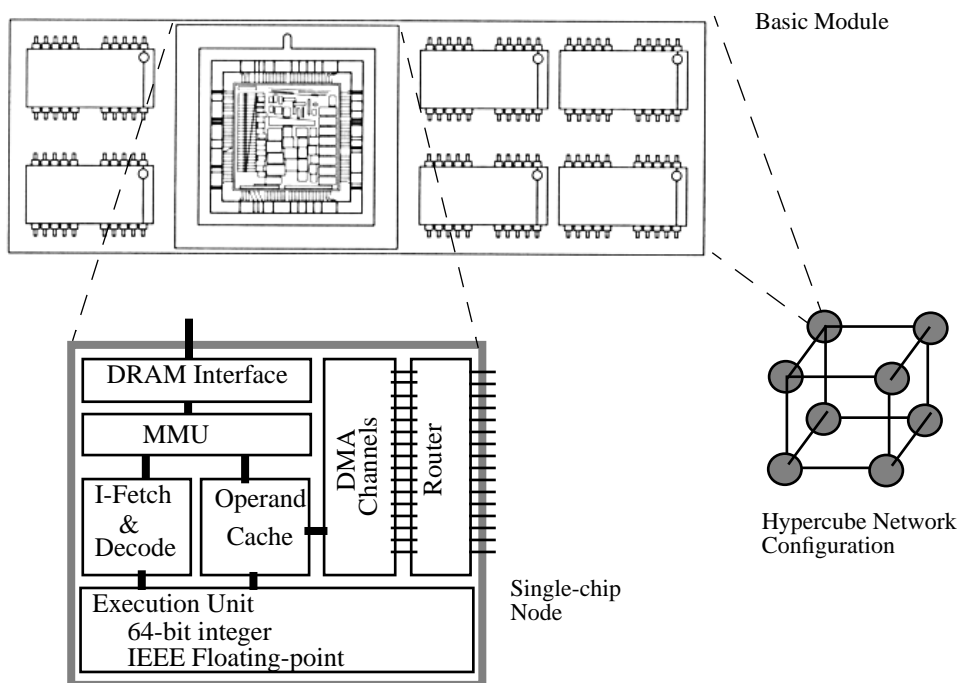
Figure 7-4 nCUBE/2 Machine Organization

The design is based on a compact module comprising a single-chip node , containing the processor, memory interface, network switch, and network interface, directly connected to DRAM chips and to other nodes.

The nCUBE/2 should be understood as a design at a point in time. The node-chip contained roughly 500,000 transistor, which was large for its time. The processor was something of a "reduced VAX" running at 20 MHz with 64-bit integer operation and 64-bit IEEE Floating Point, with a peak of 7.5 MIPS and 2.4 MFLOPS double precision. The communication support essentially occupied the silicon area that would have been devoted to cache in a uniprocessor design of the same generation; the instruction cache was only 128 bytes and the data cache held eight 64-bit operands. Network links were one bit wide and the DMA channels operated at 2.22 MB/s each.

In terms of latency scaling, the nCUBE/2 is a little more complicated than our cut-through model. A message may be an arbitrary number of 32-bit words to an arbitrary destination, with the first word being the physical address of the destination node. The message is routed to the destination as a sequence of 36 bit chunks, 32 data plus 4 bits of parity, with a routing delay of 44 cycles (2,200 ns) per hop and a transfer time of 36 cycles per word. The maximum number of hops with n nodes is $\log n$ and the average distance is half that amount. In contrast, the J-

machine organized the nodes in a three dimensional grid (actually a 3D torus) so each node is connected to six neighbors with very short wires. Individual links were relatively wide, eight bits. The maximum number of hops in this organization is roughly $\frac{3}{2}\sqrt[3]{n}$ and the average is half this many.

Board-level integration

The most common hardware design strategy for large scale machines obtains a moderately dense packing by using standard microprocessor components and integrating them at the board level. Representatives of this approach include the CalTech "hypercube" machines[Sei85], the Intel iPSC and iPSC/2, which essentially placed the core of an early personal computer on each node. The Thinking Machines CM-5 replicated the core of a Sun SparcStation 1 workstation, the CM-500 replicated the Sparcstation 10, and the Cray T3D and T3E essentially replicate the core of a DEC Alpha workstation. Most recent machines place a few processors on a board, and in some cases each board is bus-based multiprocessor. For example, the Intel ASCI RED machine is more than four thousand Pentium Pro two-way multiprocessors.

The Thinking Machine CM-5 is a good representative of this approach, circa 1993. The basic hardware organization of the CM-5 is shown in Figure 7-12. The node comprised essentially the components of contemporary workstation, in this case a 33 MHz Sparc microprocessor, its external floating-point unit, and cache controller connected to an MBUS-based memory system.[1] The network interface was an additional ASIC on the Sparc MBUS. Each node connected to two data networks, a control network, and a diagnostic network. The network is structured as a 4-ary tree with the processing nodes at the leaves. A board contains four nodes and a network switch that connects these nodes together at the first level of network. In order to provide scalable bandwidth, the CM-5 uses a kind of multi-rooted tree, called a Fat-Tree (discussed in Chapter 7), which has the same number of network switches at each level. Each board contains one of the four network switches forming the second level of the network for sixteen nodes. Higher levels of the network tree reside on additional network boards, which are cabled together. Several boards fit in a rack, but for configurations on the scale of a thousand nodes several racks are cabled together using large wiring bundles. In addition, racks of routers are used to complete the interconnection network. The links in the network are 4 bits wide, clocked at 40 MHz, delivering a peak bandwidth of about 12 MB/s. The routing delay is 10 cycles per hop, with at most $2\log_4 n$ hops.

The CM-5 network provided a kind of scalable backplane, supporting multiple independent user partitions, as well as a collection of I/O devices. Although memory is distributed over the processors, a dancehall approach is adopted for I/O. A collection of dedicated I/O nodes are accessed uniformly across the network from the processing nodes. Other machines employing similar board-level integration, such as the Intel Paragon and Cray T3D, T3E, connect the boards in a grid-like fashion to keep the wires short and use wider, faster links[Dal90]. I/O nodes are typically on the faces of the grid, or occupy internal "planes" of the cube.

---

1. The CM-5 used custom memory controllers which contain a dedicated, memory mapped vector accelerator. This aspect of the design grew out of the CM-2 SIMD heritage of the machine, and is incidental to the physical machine scaling.
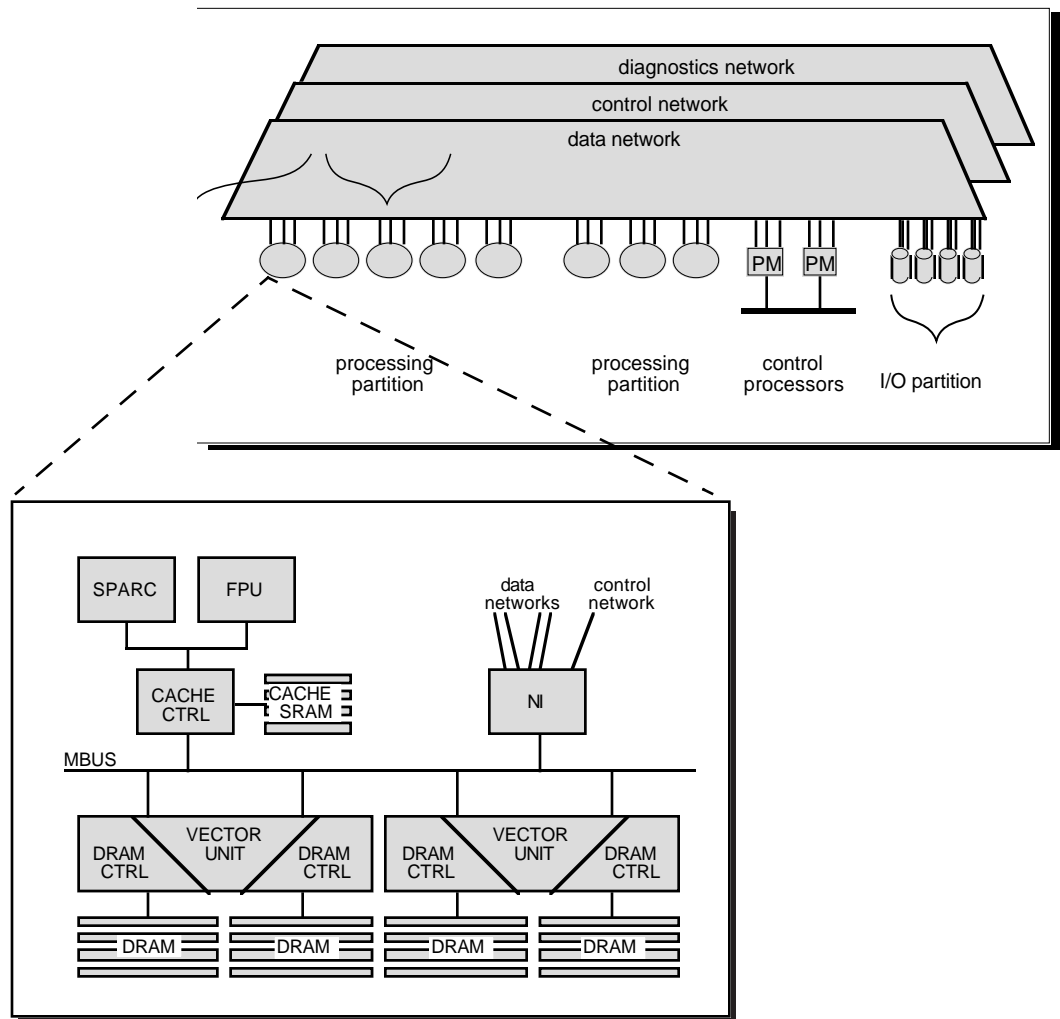
Figure 7-5 CM-5 machine organization

Each node is a repackaged SparcStation chipset (processor, FPU, MMU, Cache, Memory controller and DRAM) with a network interface chip on the MBus. The networks (data, control, and diagnostic) form a "scalable backplane" connecting computational partitions with I/O nodes.

System-level integration

Some recent large-scale machines employ much less dense packaging in order to reduce the engineering time to utilize new microprocessor and operating system technology. The IBM Scalable Parallel systems (SP-1 and SP-2) is a good representative; it puts several almost complete RS/6000 workstations into a rack. Since a complete, standard system is used for the node, the communication assist and network interface are part of a card that plugs into the system. For the IBM SPs this is a Microchannel adapter card connecting to a switch in the base of the rack. The individual network links operate at 40 MB/s. Eight to sixteen nodes fill a rack, so large configurations are built by cabling together a number of racks, including additional router racks. With a com-

plete system at every node, there is the option of distributed disks and other I/O devices over the entire machine. In the SP systems, the disks on compute nodes are typically used only for swap space and temporary storage. Most of the I/O devices are concentrated on dedicated I/O nodes. This style of design allows for a degree of heterogeneity among the nodes, since all that is required is that the network interface card can be inserted. For example, the SP systems support several models of workstations as nodes, including SMP nodes.

The high-speed networking technology developed for large-scale parallel machines has migrated down into a number of widely used local area networks (LANs). For example, ATM (asynchronous transfer mode) is a scalable, switch-based network supporting 155 Mb/s and 622 Mb/s links. FDDI switches connecting many 100 Mb/s rings are available, along with switch based fiberchannels and HPPI. Many vendors support switch-based 100 Mb/s Ethernet and switch-based gigabit Ethernet is emerging. In addition, a number of higher bandwidth, lower latency system area networks (SAN), which operate over shorter physical distances, have been commercialized, including Myrinet, SCI, and SeverNet. These networks are very similar to traditional large-scale multiprocessor networks, and allow conventional computer systems to be integrated into a "cluster" with a scalable, low latency interconnect. In many cases, the individual machines are small scale multiprocessors. Because of the nodes are complete, independent systems, this approach is widely used to provide high availability services, such as data bases, where if one node fails its job "fails-over" to the others.

7.2.5 Scaling in a Generic Parallel Architecture

The engineering challenges of large-scale parallel machines are well understood today, with several companies routinely producing systems of several hundred to a thousand high-performance microprocessors. The tension between tighter physical integration and engineering time to incorporate new microprocessor technology into large-scale machines gives rise to a wide spectrum of packaging solutions, all of which have the conceptual organization of our generic parallel architecture in Figure 7-2. Scalable interconnection network design is an important and interesting sub-area of parallel architecture which has advanced dramatically over recent years, as we will see in Chapter 10. Several "good" networks have been produced commercially, which offer high per link bandwidth and reasonably low latency that increases slowly with the size of the system. Furthermore, these networks provide scalable aggregate bandwidth, allowing a very large number of transfers to occur simultaneously. They are also robust, in that the hardware error rates are very low, in some cases as low as modern busses. Each of these designs have some natural scaling limit as a result of bandwidth, latency, and cost factors, but from an engineering viewpoint it is practical today to build machines on a scale that is limited primarily by financial concerns.

In practice, the target maximum scale is quite important in assessing design trade-offs. It determines the level of design and engineering effort warranted by each aspect of the system. For example, the engineering required to achieve the high packaging density and degree of modularity needed to construct very large systems may not be cost-effective at the moderate scale where less sophisticated solutions suffice. A practical design seeks a balance between computational performance, communication performance, and cost at the time when the machine is produced. For example, better communication performance or better physical density might be achieved by integrating the network more closely with the processor. However, this may increase cost or compromise performance, either by increasing the latency to memory or increasing design time and thus might not be the most effective choice. With processor performance improving rapidly over time, a more rudimentary design starting later on the technology curve might be produced at the

same time with higher computational performance, but perhaps less communication performance. As with all aspects of design, it is a question of balance.

What the entire spectrum of large-scale parallel machines have in common is that there can be a very large number of transfers on-going simultaneously, there is essentially no instantaneous global information or global arbitration, and the bulk of the communication time is attributable to the node-to-network interface. These are the issues that dominate our thinking from an architectural viewpoint. It is not enough that the hardware capability scales, the entire system solution must scale, including the protocols used to realize programming models and the capabilities provided by the operating system, such as process scheduling, storage management, and I/O. Serialization due to contention for locks and shared resources within applications or even the operating system may limit the useful scaling of the system, even if the hardware scales well in isolation. Given that we have met the engineering requirements to physically scale the system to the size of interest in a cost-effective manner, we must also ensure that the communication and synchronization operations required to support the target programming models scale and have a sufficiently small fixed cost to be effective.

7.3 Realizing Programming Models

In this section we examine what is required to implement programming models on large distributed memory machines. Historically, these machine have been most strongly associated with message passing programming models, but shared address space programming models have become increasingly important and well represented. Chapter 1 introduced the concept of a communication abstraction, which defined the set of communication primitives provided to the user. These could be realized directly in the hardware, via system software, or through some combination of the two, as illustrated by Figure 7-6. This perspective focuses our attention on the extensions of the computer architecture of the node which support communication. In our study of small scale shared memory machines, we examined a class of designs where the communication abstraction is supported directly in hardware as an extension of the memory interface. Data is shared and communicated by accesses to shared variables, which translate into load, store, or atomic update instructions on virtual addresses that are mapped to the same physical memory address. To obtain reasonable latency and bandwidth on accesses to shared locations, it was essential that shared locations could be replicated into caches local to the processors. The cache controllers were extended to snoop and respond to bus transactions, as well as processor requests, and the bus was extended to support detection of sharing. The load and store operations in the coherent shared memory abstraction were constructed as a sequence of primitive bus transactions according to a specific protocol defined by a collection of state machines.

In large scale parallel machines, the programming model is realized in a similar manner, except the primitive events are transactions across the network, *i.e.,* network transactions rather than bus transactions. *A network transaction is a one-way transfer of information from an output buffer at the source to an input buffer at the destination causing some kind of action to occur at the destination, the occurrence of which is not directly visible at the source.* This is illustrated in Figure 7-7. The action may be quite simple, e.g., depositing the data into an accessible location or making transition in a finite-state machine, or it can be more general, e.g., execution of a message handler routine. The effects of a network transaction are observable only through additional transactions. Traditionally, the communication abstraction supported directly by the hardware was usually hid-

Parallel Applications

Multiprogramming   Shared   Message   Data
                   Address   Passing   Parallel   *Programming Models*

                        Compilation
                           or   Communication Abstraction
                         Library   User/System Boundary
                                     Operating Systems Support
                                                              Hardware/Software Boundary
                        Communication Hardware

                      Physical Communication Medium
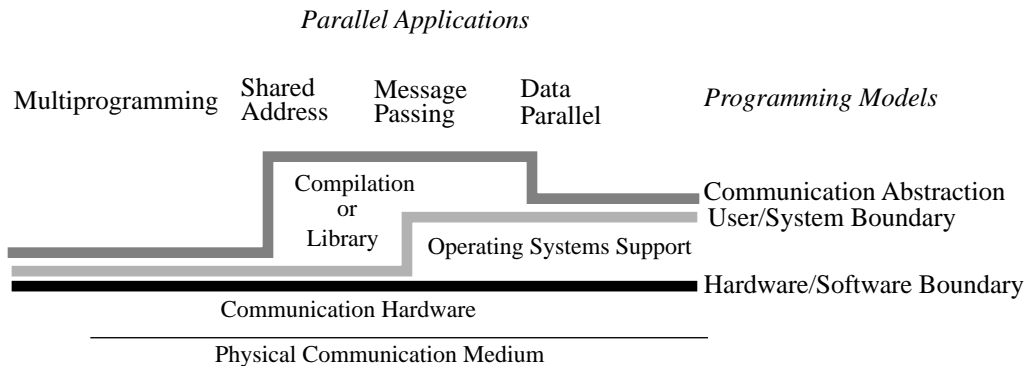
Figure 7-6 Layers of Parallel Archtecture

The figure illustrates the critical layers of abstractions and the aspects of the system design that realize each of the layers.

den below the vendors message passing library, but increasingly the lower level abstraction is accessible to user applications.

The differences between bus and network transactions have far reaching ramifications. The potential design space is much larger than what we saw in Chapter 5, where the bus provided serialization and broadcast, and the primitive events were small variations on conventional bus transactions. In large scale machines there is tremendous variation in the primitive network transaction itself, as well as how these transactions are driven and interpreted at the end points. They may be presented to the processor as I/O operations and driven entirely by software, or they may be integrated into the memory system and driven by a dedicated hardware controller. A very wide spectrum of large-scale machines have been designed and implemented, emphasizing a range of programming models, employing a range of primitives at the hardware/software boundary, and providing widely differing degrees of direct hardware support and system software intervention.

To make sense of this great diversity, we proceed step-by-step. In this section, we first define a network transaction more precisely and contrast it more carefully with a bus transaction. We then sketch what is involved in realizing shared memory and message passing abstractions out of this primitive, without getting encumbered by the myriad of ways that a network transaction itself might be realized. In later sections we work systematically through the space of design options for realizing network transactions and the programming models built upon them.

7.3.1 Primitive Network Transactions

To understand what is involved in a network transaction, let us first reexamine what is involved in a basic bus transaction in some detail, since similar issues arise. Before starting a bus transaction, a protection check has been performed as part of the virtual-to-physical address translation. The *format* of information in a bus transaction is determined by the physical wires of the bus, e.g., the data lines, address lines, and command lines. The information to be transferred onto the bus is held in special *output registers*, e.g., an address, command, and data registers, until it can be driven onto the bus. A bus transaction begins with *arbitration* for the medium. Most busses employ a global arbitration scheme where a processor requesting a transaction asserts a bus-
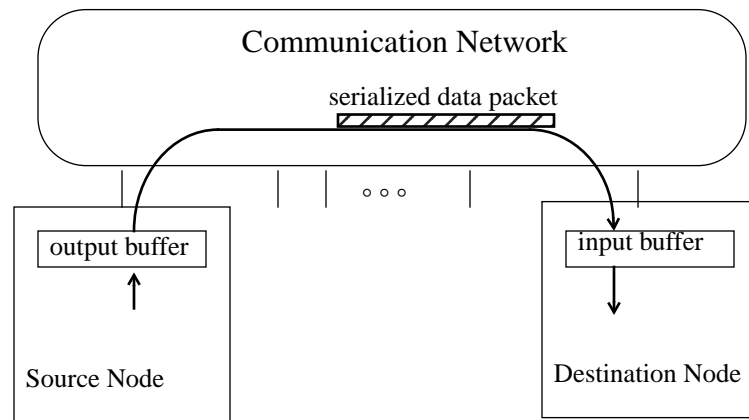
Figure 7-7 Network Transaction Primitive

A one-way transfer for information from a source output buffer to an input buffer of a designated destination, causing some action to take place at the destination.

request line and waits for the corresponding bus-grant. The *destination* of the transaction is implicit in the address. Each module on the bus is configured to respond to a set of physical addresses. All modules examine the address and one responds to the transaction. (If none responds the bus controller detects the time-out and aborts the transaction.) Each module includes a set of *input registers*, capable of buffering any request to which it might respond. Each bus transaction involves a *request* followed by a *response*. In the case of a read, the response is the data and an associated completion signal; for a write it is just the completion acknowledgment. In either case, both the source and destination are informed of the *completion* of the transaction. In many busses, each transaction is guaranteed to complete according to a well-defined schedule. The primary variation is the length of time that it takes for the destination to turn around the response. In split-transaction busses, the response phase of the transaction may require rearbitration and may be performed in a different order than the requests.

Care is required to avoid deadlock with split-transactions (and coherence protocols involving multiple bus transactions per operation) because a module on the bus may be both requesting and servicing transactions. The module must continue servicing bus requests and sink replies, while it is attempting to present its own request. One wants to ensure that for any transaction that might be placed on the bus, sufficient input buffering exists to accept the transaction at the destination. This can be addressed in a conservative fashion by providing enough resources for the worst case, or optimistically with the addition of a negative acknowledgment signal (NAK). In either case, the solution is relatively straight-forward because there are *few concurrent communication operations* that can be in progress on a bus, and the source and destination are directly coupled by a wire.

The discussion of busses raises the issues present in a network transaction as well. These are:

- protection
- format
- output buffering
- media arbitration

- destination name and routing
- input buffering
- action
- transaction ordering
- completion detection
- transaction ordering
- deadlock avoidance
- delivery guarantee

The fundamental difference is that the source and destination of the transaction are uncoupled, *i.e.,* there may be no direct wires between them, and there is no global arbitration for the resources in the system. There is no global information available to all at the same instant, and there may be a huge number of transactions in progress simultaneously. These basic differences give each of the issues listed above a very different character than in a bus. Let us consider each in turn.

Protection: As the number of components becomes larger, the coupling between components looser, and the individual components more complex, it may be worthwhile to limit the how much each component trusts the others to operate correctly. Whereas in a bus-based system all protection checks are performed by the processor before placing a transaction on the bus, in a scalable system, individual components will often perform checks on the network transaction, so that an errant program or faulty hardware component cannot corrupt other components of the system.

Format: Most network links are narrow, so the information associated with a transaction is transferred as a serial stream. Typical links are a few (1 to 16) bits wide.The format of the transaction is dictated by how the information is serialized onto the link, unlike in a bus where it is a parallel transfer whose format is determined by the physical wires. Thus, there is a great deal of flexibility in this aspect of design. One can think of the information in a network transaction as an envelope with more information inside. The envelope includes information germane to the physical network to get the packet from its source to its destination port. This is very much like the command and address portion of a bus transaction, which tells all parties involved what to do with the transaction. Some networks are designed to deliver only fixed sized packets, others can deliver variable size packets. Very often the envelop contains additional envelops within it. The communication assist may wrap up the user information in an envelop germane to the remote communication assist, and put this within the physical network envelop.

Output buffering: The source must provide storage to hold information that is to be serialized onto the link, either in registers, FIFOs, or memory. If the transaction is of fixed format, this may be as simple as the output buffer for a bus. Since network transactions are one-way and can potentially be pipelined, it may be desirable to provide a queue of such output registers. If the packet format is variable upto some moderate size, a similar approach may be adopted where each entry in the output buffer is of variable size. If a packet can be quite long, then typically the output controller contains a buffer of descriptors, pointing to the data in memory. It then stages portions of the packet from memory into small output buffers and onto the link, often through DMA transfer.

Media arbitration: There is no global arbitration for access to the network and many network transactions can be initiated simultaneously. (In bus-like networks, such as ethernet or rings there is distributed arbitration for the single or small number of transactions that can occur simultaneously.) Initiation of the network transaction places an implicit claim on resources in the communication path from the source to the destination, as well as resources at the destination. These resources are potentially shared with other transactions. Local arbitration is performed at the source to determine whether or not to initiate the transaction. However, this usually does not imply that all necessary resources are reserved to the destination; the resources are allocated incrementally as the message moves forward.

Destination name and routing: The source must be able to specify enough information to cause the transaction to be routed to the appropriate destination. This is in contrast to the bus, where the source simply places the address on the wire and the destination chooses whether it should accept the request. There are many variations in how routing is specified and performed, but basically the source performs a translation from some logical name for the destination to some form of physical address.

Input buffering: At the destination the information in the network transaction must be transferred from the physical link into some storage element. As with the output buffer, this may be simple registers, a queue, or it may be delivered directly into memory. The key difference is that transactions may arrive from many sources; in contrast, the source has complete control over how many transactions it initiates. The input buffer is in some sense a shared resource used by many remote processors; how this is managed and what happens when it fills up is a critical issue that we will examine later.

Action: The action taken at the destination may be very simple, say a memory access, or it may be complex. In either case it may involve initiating a response.

Completion detection: The source has indication that the transaction has been delivered into the network, but usually no indication that it has arrived at its destination. This completion must be inferred from a response, an acknowledgment, or some additional transaction.

Transaction ordering: Whereas a bus provides strong ordering properties among transactions, in a network the ordering is quite weak. Even on a split-transaction bus with multiple outstanding transactions, we could rely on the serial arbitration for the address bus to provide a global order. Some networks will ensure that a sequence of transactions from a given source to a single destination will be seen in order at the destination, others will not even provide this very limited assurance. In either case, no node can perceive the global order. In realizing programming models on large-scale machines, ordering constraints must be imposed through network transactions.

Deadlock avoidance: Most modern networks are deadlock free as long as the modules on the network continue to accept transactions. Within the network this may require restrictions on permissible routes or other special precautions, as we discuss in Chapter 10. Still we need to be careful that our use of network transactions to realize programming models does not introduce deadlock. In particular, while we are waiting, unable to source a transaction, we usually will need to continue accepting in-coming transactions. This situation is very much like that with split-transaction busses, except that the number of simultaneous transactions is much larger and there is no global arbitration or immediate feedback.

Delivery Guarantees: A fundamental decision in the design of a scalable network is the behavior when the destination buffer is full. This is clearly an issue on an end-to-end basis, since it is non-trivial for the source to know whether the destination input buffer is available when it is attempting to initiate a transaction. It is also an issue on a link-by-link basis within the network itself. There are two basic options: discard information if the buffer is full or defer transmission until space is available. The first requires that there be a way to detect the situation and retry, the second requires a flow-control mechanism and can cause the transactions to back-up. We will examine both options in this chapter.

In summary, a network transaction is a one-way transfer for information from a source output buffer to an input buffer of a designated destination, causing some action to take place at the destination. Let's consider what is involved in realizing the communication abstractions found in common programming models in terms of this primitive.

7.3.2 Shared Address Space

Realizing the shared address space communication abstraction fundamentally requires a two-way request response protocol, as illustrated abstractly in Figure 7-8. A global address is decomposed into a module number and a local address. For a read operation, a request is sent to the designated module requesting a load of the desired address and specifying enough information to allow the result to be returned to the requestor through a response network transaction. A write is similar, except that the data is conveyed with the address and command to the designated module and the response is merely an acknowledgment to the requestor that the write has been performed. The response informs the source that the request has been received or serviced, depending on whether it is generated before or after the remote action. The response is essential to enforce proper ordering of transactions.

A read request typically has a simple fixed format, describing the address to read and the return information. The write acknowledgment is also simple. If only fixed sized transfers are supported, *e.g.,* a word or a cache block, then the read response and write request are also of simple fixed format. This is easily extended to support partial word transfers, say by including byte enables, whereas arbitrarily length transfers require a more general format. For fixed-format transfers, the output buffering is typically as with a bus. The address, data, and command are staged in an output register and serialized onto the link.

The destination name is generally determined as result of the address translation process which converts a global address to a module name, or possibly a route to the module, and an address local to that module. Succeeding with the translation usually implies authorization to access the designated destination module, however, the source must still gain access to the physical network and the input buffer at the remote end. Since a large number of nodes may issue requests to the same destination and there is no global arbitration or direct coupling between the source and destination, the combined storage requirement of the requests may exceed the input buffering at the node. The rate at which the requests can be processed is merely that of a single node, so the requests may back up through the network, perhaps even to the sources. Alternatively, the requests might be dropped in this situation, requiring some mechanism for retrying. Since the network may not be able to accept a request when the node attempts to issue it, each node must be able to accept replies and requests, even while unable to inject its own request, so that the packets in the network can move forward. This is the more general form of the fetch deadlock issue observed in the previous chapter. This *input buffer problem* and the *fetch deadlock problem*

Source   Destination

(1) Initiate memory access   load r <– [global address]
(2) Address Translation
(3) Local / Remote Check

(4) Request Transaction
```
read request
```
```
read request
```

(5) Remote Memory Access
wait
memory access

```
read response
```

(6) Reply Transaction
```
read response
```

(7) Complete memory access

Time
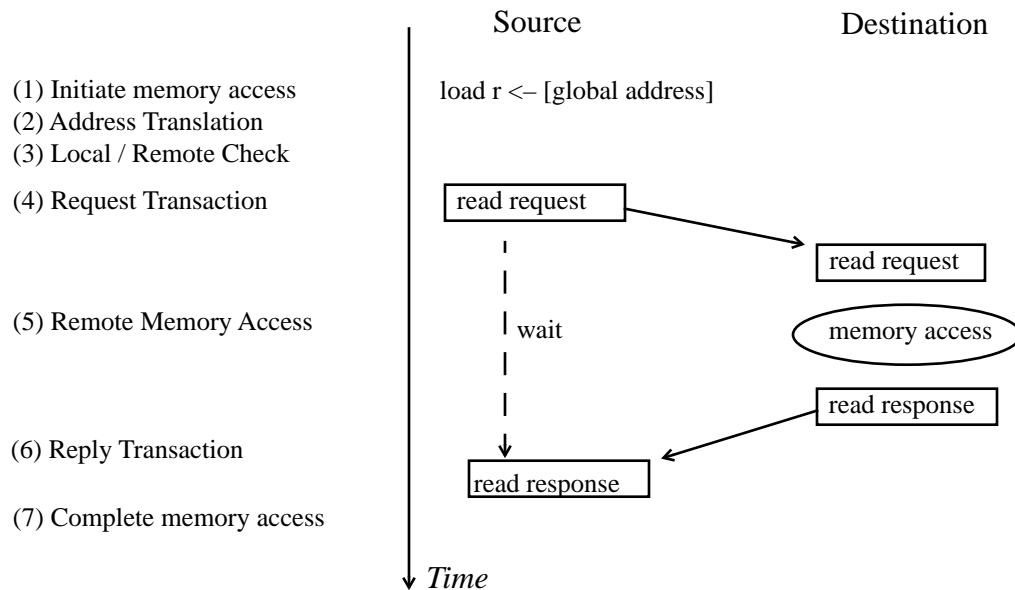
Figure 7-8 Shared Address Space Communication Abstraction

The figure illustrates the anatomy of a read operation in large scale machine in terms of primitive network transactions. (1) The source processor initiates the memory access on global address V. The global address is translated into a node number or route a local address on that node: <P, A>. (3) A check is performed to determine if the address is local to the issuing processor. (4) If not, a read request transaction is performed to deliver the request to the designated processor, which (5) accesses the specified address, and (6) returns the value in a reply transaction to the original node, which (7) completes the memory access.

arise with many different communication abstractions, so they will addressed in more detail after looking at the corresponding protocols for message passing abstractions.

When supporting a shared address space abstraction, we need to ask whether it is coherent and what memory consistency model it supports. In this chapter we consider designs that do not replicate data automatically through caches; all of the next chapter is devoted to that topic. Thus, each remote read and write go to the node hosting the address and access the location. Thus, coherence is met by the natural serialization involved in going through the network and accessing memory. One important subtlety is that the accesses from remote nodes need to be coherent with access from the local node. Thus, if shared data is cached locally, processing the remote reference needs to be cache coherent within the node.

Achieving sequential consistency in scalable machines is more challenging than in bus-based designs because the interconnect does not serialize memory accesses to locations on different nodes. Furthermore, since the latencies of network transactions tend to be large, we are tempted to try to hide this cost whenever possible. In particular, it is very tempting to issue multiple write transactions without waiting for the completion acknowledgments to come back in between. To see how this can undermine the consistency model, consider our familiar flag-based code fragment executing on a multiprocessor with physically distributed memory but no caches. The variables A and flag are allocated in two different processing nodes, as shown in Figure 7-9(a). Because of delays in the network, processor P2 may see the stores to A and flag in the reverse of the order they are generated. Ensuring point-to-point ordering among packets between each

pair of nodes does not remedy the situation, because multiple pairs of nodes may be involved. A situation with a possible reordering due to the use of different paths within the network is illustrated in Figure 7-9(b). Overcoming this problem is one of the reasons why writes need to be acknowledged. A correct implementation of this construct will wait for the write of A to be completed before issuing the write of flag. By using the completion transactions for the write, and the read response, it is straight-forward to meet the sufficient conditions for sequential consistency. The deeper design question is how to meet these conditions while minimizing the amount of waiting by determining that the write has been committed and appears to all processors as if it had performed.
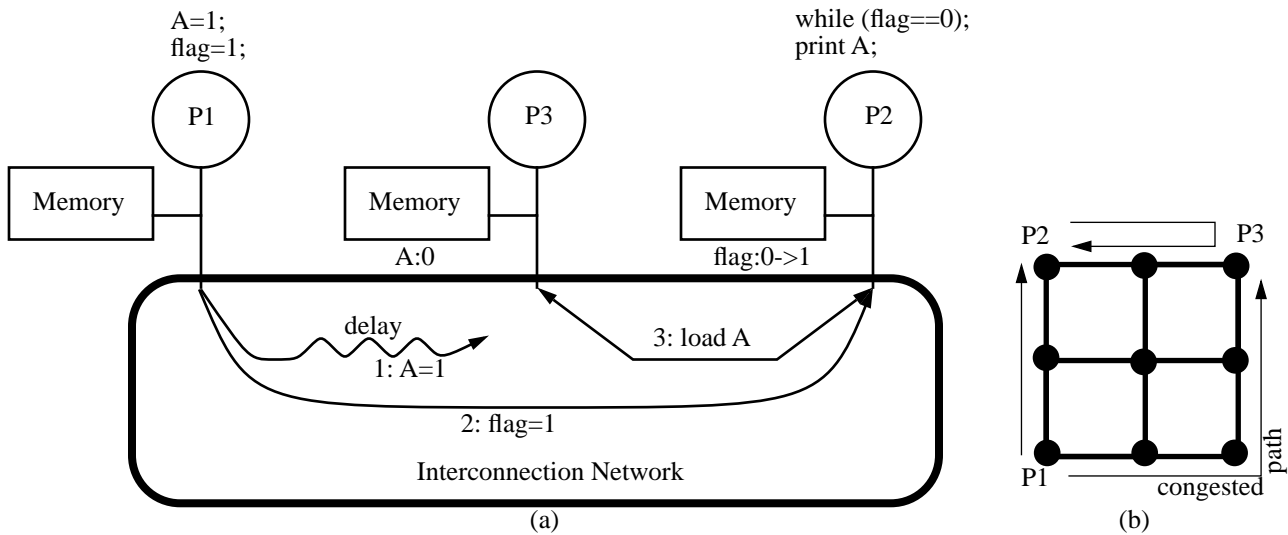
Figure 7-9 Possible reordering of memory references for shared flags

The network is assumed to preserve point-to-point order. The processors have no cache as shown in part (a). The variable A is assumed to be allocated out of P2's memory, while variable flag is assumed to be allocated from P3's memory. It is assumed that the processors do not stall on a store instruction, as is true for most uniprocessors. It is easy to see that if there is network congestion along the links from P1 to P2, P3 can get the updated value of flag from P1 and then read the stale value of A (A=0) from P2. This situation can easily occur, as indicated in part (b) where messages are always routed first in X-dimension and then in Y-dimension of a 2D mesh.

7.3.3 Message Passing

A send/receive pair in the message passing model is conceptually a one-way transfer from a source area specified by the source user process to a destination address specified by the destination user process. In addition, it embodies a pairwise synchronization event between the two processes. In Chapter 2, we noted the important semantic variations on the basic message passing abstractions, such as synchronous and asynchronous message send. User level message passing models are implemented in terms of primitive network transactions, and the different synchronization semantics have quite different implementations, i.e., different *network transaction protocols*. In most early large scale machines, these protocols were buried within the vendor kernel and library software. In more modern machines the primitive transactions are exposed to allow a wider set of programming models to be supported.

This chapter uses the concepts and terminology associated with MPI. Recall that MPI distinguishes the notion of when a call to a send or receive function returns from when message operation *completes*. A synchronous send completes once the matching receive has executed, the source data buffer can be reused, and the data is ensured of arriving in the destination receive buffer. A buffered send completes as soon as the source data buffer can be reused, independent of whether the matching receive has been issued; the data may have been transmitted or it may be buffered somewhere in the system.[1] Buffered send completion is asynchronous with respect to the receiver process. A receive completes when the message data is present in the receive destination buffer. A blocking function, send or receive, returns only after the message operation completes. A non-blocking function returns immediately, regardless of message completion, and additional calls to a probe function are used to detect completion. The protocols are concerned only with message operation and completion, regardless of whether the functions are blocking.

To understand the mapping from user message passing operations to machine network transaction primitives, let us consider first synchronous messages. The only way for the processor hosting the source process to know whether the matching receive has executed is for that information to be conveyed by an explicit transaction. Thus, the synchronous message operation can be realized with a three phase protocol of network transactions, as shown in Figure 7-10. This protocol is for a sender-initiated transfer. The send operation causes a "ready-to-send" to be transmitted to the destination, carrying the source process and tag information. The sender then waits until a corresponding "ready-to-receive" has arrived. The remote action is to check a local table to determine if a matching receive has been performed. If not, the ready-to-send information is recorded in the table to await the matching receive. If a matching receive is found, a "ready-to-receive" response transaction is generated. The receive operation checks the same table. If a matching send is not recorded there, the receive is recorded, including the destination data address. If a matching send is present, the receive generates a "ready-to-receive" transaction. When a "ready-to-receive" arrives at the sender, it can initiate the data transfer. Assuming the network is reliable, the send operation can complete once all the data has been transmitted. The receive will complete once it all arrives. Note that with this protocol both the source and destination nodes know the local addresses for the source and destination buffers at the time the actual data transfer occurs. The 'ready' transactions are small, fixed format packets, whereas the data is a variable length transfer.

In many message passing systems the matching rule associated with a synchronous message is quite restrictive and the receive specifies the sending process explicitly. This allows for an alternative "receiver initiated" protocol in which the match table is maintained at the sender and only two network transactions are required (receive ready and data transfer).

---

1. The standard MPI mode is a combination of buffered and synchronous that gives the implementation substantial freedom and the programmer few guarantees. The implementation is free to choose to buffer that data, but cannot be assumed to do so. Thus, when the send completes the send buffer can be reused, but it cannot be assumed that the receiver has reached the point of the receive call. Nor can it be assumed that send buffering will break the deadlock associated with two nodes sending to each other and then posting the receive. Non-blocking sends can be used to avoid the deadlock, even with synchronous sends.

The read-mode send is a stronger variant of synchronous mode, where it is an error if the received has not executed. Since the only to obtain knowledge of the state of the non-local processes is through exchange of messages, an explicit message event would need to be used to indicate readiness. The race condition between posting the ready receive and transmitting the synchronization message is very similar to the flags example in the shared address space case.
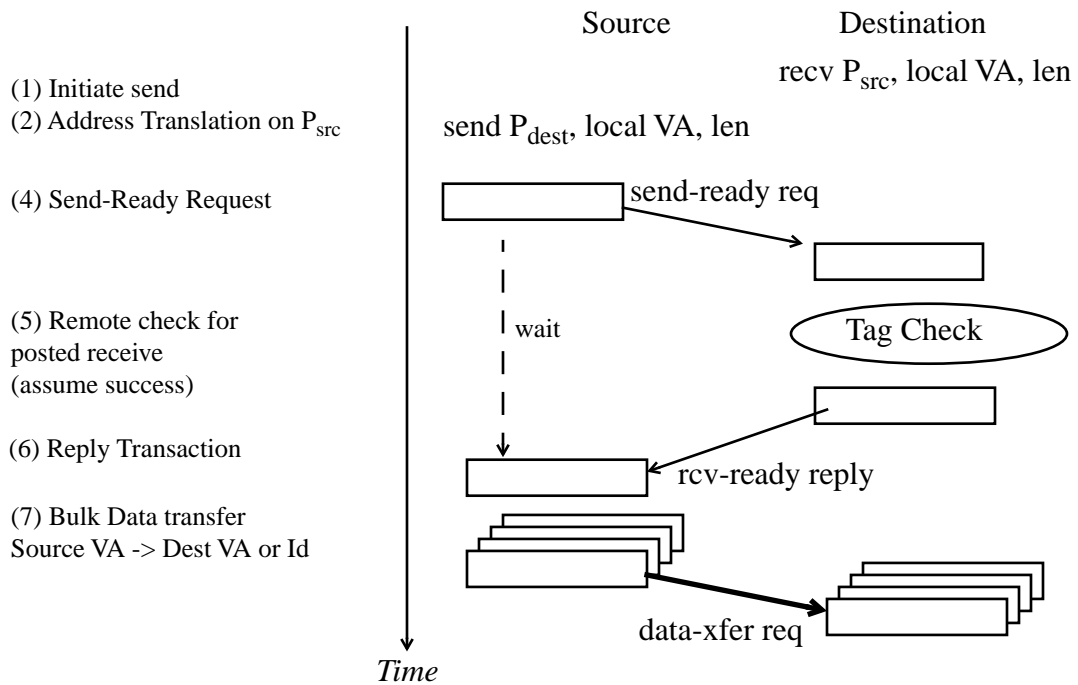
Figure 7-10 Synchronous Message Passing Protocol

The figure illustrates the three-way handshake involved in realizing a synchronous send/receive pair in terms of primitive network transactions.

The buffered send is naively implemented with an optimistic single-phase protocol, as suggested in Figure 7-11. The send operation transfers the source data in a single large transaction with an envelop containing the information used in matching, e.g., source process and tag, as well as length information. The destination strips off the envelop and examines its internal table to determine if a matching receive has been posted. If so, it can deliver the data at the specified receive address. If no matching receive has been posted, the destination allocates storage for the entire message and receives it into the temporary buffer. When the matching receive is posted later, the message is copied to the desired destination area and the buffer is freed.

This simple protocol presents a family of problems. First, the proper destination of the message data cannot be determined until after examining the process and tag information and consulting the match table. These are fairly expensive operations, typically performed in software. Meanwhile the message data is streaming in from the network at high rate. One approach is to always receive the data into a temporary input buffer and then copy it to its proper destination. Of course, this introduces a store-and-forward delay and, possibly, consumes a fair amount of processing resources.

The second problem with this optimistic approach is analogous to the input buffer problem discussed for a shared address space abstraction since there is no ready-to-receive handshaking before the data is transferred. In fact, the problem is amplified in several respects. First, the transfers are larger, so the total volume of storage needed at the destination is potentially quite large.

Second, the amount of buffer storage depends on the program behavior; it is not just a result of the rate mismatch between multiple senders and one receiver, much less a timing mismatch where the data happens to arrive just before the receiver was ready. Several processes may choose to send many messages each to a single process, which happens not to receive them until much later. Conceptually, the asynchronous message passing model assumes an unbounded amount of storage *outside* the usual program data structures where data is stored until the receives for them are posted and performed. Furthermore, the blocking asynchronous send must be allowed to complete to avoid deadlock in simple communication patterns, such as a pairwise exchange. The program needs to continue executing past the send to reach a point where a receive is posted. Our optimistic protocol does not distinguish transient receive-side buffering from the prolonged accumulation of data in the message passing layer.
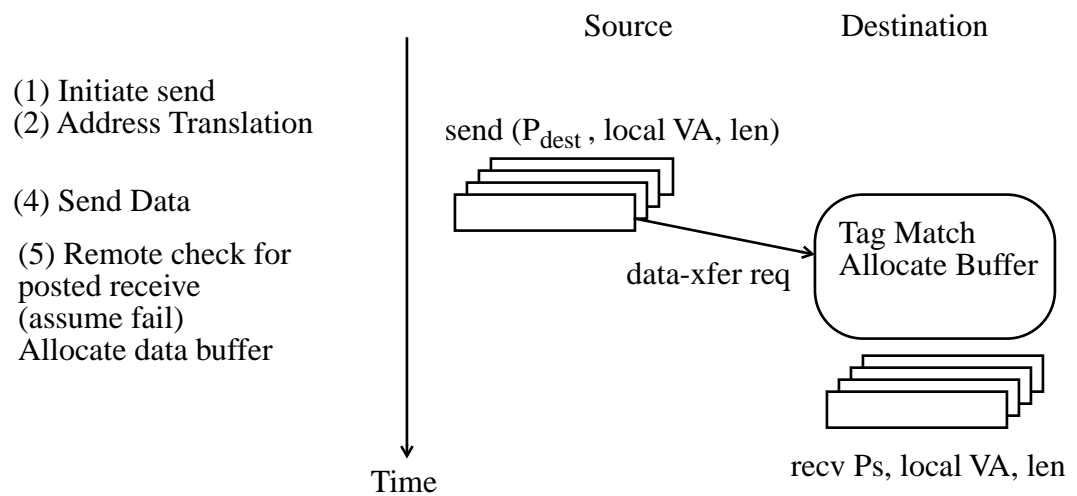
Figure 7-11 Asynchronous (optimistic) Message Passing Protocol

The figure illustrates a naive single-phase optimisitic protocol for asynchronous message passing where the source simply delivers that data to the destination, without concern for whether the destination has storage to hold it.

More robust message passing systems use a three phase protocol for long transfers. The send issues a ready-to-send with the envelop, but keeps the data buffered at the sender until the destination can accept it. The destination issues a ready-to-receive either when it has sufficient buffer space or a matching receive has been executed so the transfer can take place to the correct destination area. Note that in this case the source and destination addresses are known at both sides of the transfer before the actual data transfer takes place. For short messages, where the handshake would dominate the actual data transfer cost, a simple credit scheme can be used. Each process sets aside a certain amount of space for processes that might send short messages to it. When a short message is sent, the sender deducts its destination "credit" locally, until it receives notification that the short message has been received. In this way, a short message can usually be launched without waiting a round-trip delay for the handshake. The completion acknowledgment is used later to determine when additional short messages can be sent without the handshake.

As with the shared address space, there is a design issue concerning whether the source and destination addresses are physical or virtual. The virtual-to-physical mapping at each end can be per-

Source   Destination

(1) Initiate send
(2) Address Translation on Pd

send Pd, local VA, len

(4) Send-Ready Request

      send-rdy req

(5) Remote check for
posted receive
(assume fail)
Record Send-rdy

return and compute

Tag Check

(6) Rcv-ready reguest

recv Ps, local VA, len

(7) Bulk Data reply
Source VA -> Dest VA or Id

rcv-rdy req

data-xfer reply

Time
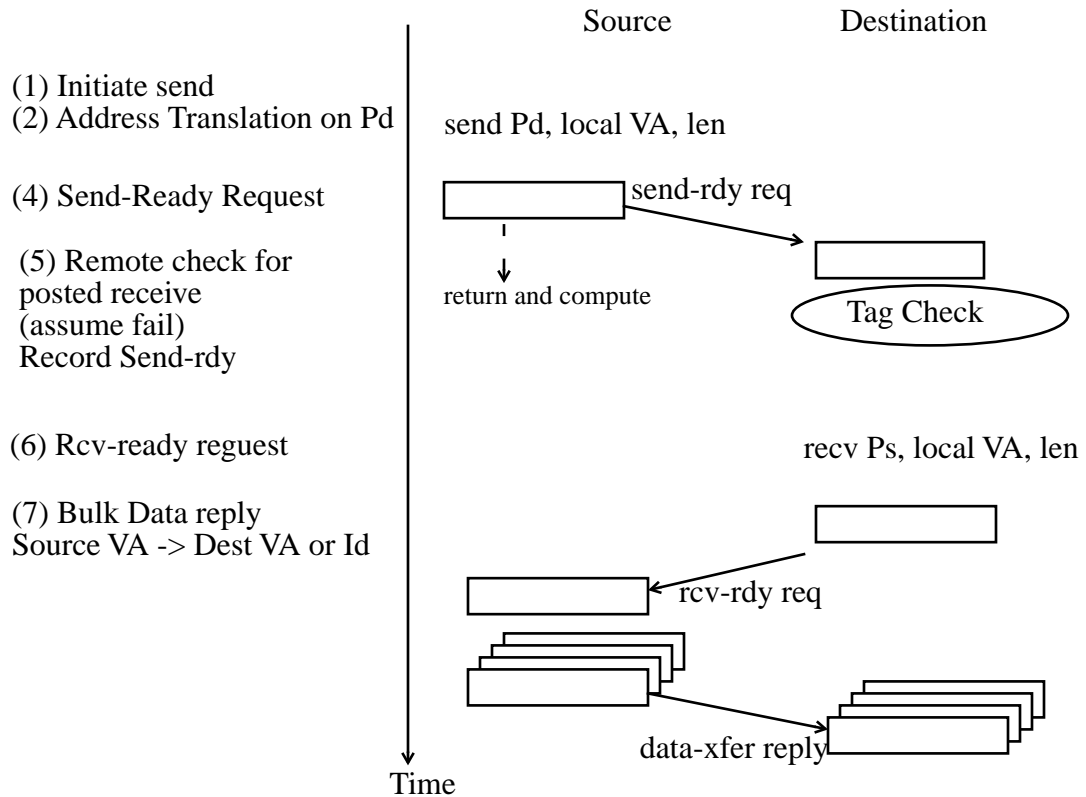
Figure 7-12 Asynchronous Conservative Message Passing Protocol

The figure illustrates a one-plus-two phase conservative protocol for asynchronous message passing. The data is held at the source until the matching receive is executed, making the destination address known before the data is delivered.

formed as part of the send and receive calls, allowing the communication assist to exchange physical addresses, which can be used for the DMA transfers. Of course, the pages must stay resident during the transfer for the data to stream into memory. However, the residency is limited in time, from just before the handshake until the transfer completes. Also, very long transfers can be segmented at the source so that few resident pages are involved. Alternatively, temporary buffers can be kept resident and the processor relied upon to copy the data from and to the source and destination areas. The resolution of this issue depends heavily on the capability of the communication assist, as discussed below.

In summary, the send/receive message passing abstraction is logically a one-way transfer, where the source and destination addresses are specified independently by the two participating processes and an arbitrary amount of data can be sent before any is received. The realization of this class of abstractions in terms of primitive network transactions typically requires a three-phase protocol to manage the buffer space on the ends, although an optimistic single-phase protocol can be employed safely to a limited degree.

7.3.4 Common challenges

The inherent challenge in realizing programming models in large scale systems is that each processor has only limited knowledge of the state of the system, a very large number of network transactions can be in progress simultaneously, and the source and destination of the transaction are decoupled. Each node must infer the relevant state of the system from its own point-to-point events. In this context, it is possible, even likely, for a collection of sources to seriously overload a destination before any of them observe the problem. Moreover, because the latencies involved are inherently large, we are tempted to use optimistic protocols and large transfers. Both of these increase the potential over-commitment of the destination. Furthermore, the protocols used to realize programming models often require multiple network transactions for an operation. All of these issues must be considered to ensure that forward progress is made in the absence of global arbitration. The issues are very similar to those encountered in bus-based designs, but the solutions cannot rely on the constraints of the bus, namely a small number of processors, a limited number of outstanding transactions, and total global ordering.

Input buffer overflow

Consider the problem of contention for the input buffer on a remote node. To keep the discussion simple, assume for the moment fixed-format transfers on a completely reliable network. The management of the input buffer is simple: a queue will suffice. Each incoming transaction is placed in the next free slot in the queue. However, it is possible for a large number of processors to make a request to the same module at once. If this module has a fixed input buffer capacity, on a large system it may become overcommitted. This situation is similar to the contention for the limited buffering within the network and it can be handled in a similar fashion. One solution is to make the input buffer large and reserve a portion of it for each source. The source must constrain its own demand when it has exhausted its allotment at the destination. The question then arises, how does the source determine that space is again available for it? There must be some flow control information transmitted back from the destination to the sender. This is a design issue that can be resolved in a number of ways: each transaction can be acknowledged, or the acknowledgment can be coupled with the protocol at a higher level. For example, the reply indicates completion of processing the request.

An alternative approach, common in reliable networks, is for the destination to simply refuse to accept incoming transactions when its input buffer is full. Of course, the data has no place to go, so it remains stuck in the network for a period of time. The network switch feeding the full buffer will be in a situation where it cannot deliver packets as fast as they are arriving. Given that it also has finite buffering, it will eventually refuse to accept packets on its inputs. This phenomena is called *back-pressure*. If the overload on the destination is sustained long enough, the backlog will build in a tree all the way back to the sources. At this point, the sources feel the back pressure from the overloaded destination and are forced to slow down such that the sum of the rates from all the sources sending to the destination are no more than what the destination can receive. One might worry that the system will fail to function in such situations. Generally, networks are built so that they are *deadlock free,* which means that messages will make forward progress as long as messages are removed from the network at the destinations[DaSe87]. (We will see in Chapter 7 that it takes care in the network design to ensure deadlock freedom. The situations that need to be avoided are similar to traffic gridlock, where all the cars in an intersection are preventing the others from moving out of the way.) So, there will be forward progress. The problem is that with the network so backed up, messages not headed for the overloaded destination will get stuck in traf-

fic also. Thus, the latency of all communication increases dramatically with the onset of this backlog.

Even if there is a large backlog, the messages will eventually move forward. Networks typically have only finite buffering per switch, so eventually the backlog will reach back to the sources and the network will refuse to accept any incoming messages. Thus, the destinations exert back-pressure through the network, which eventually reaches all the way to the sources. This situation may cause additional transactions to queue up within the network in front of the full buffers, thereby increasing the communication latency. This situation establishes an interesting "contract" between the nodes and the network. From the source point of view, if the network accepts a transaction it is guaranteed that the transaction will eventually be delivered to the destination. However, a transaction may not be accepted for an arbitrarily large period of time and during that time the source must continue to accept incoming transactions.

Alternatively, the network may be constructed so that the destination can inform the source of the state of its input buffer. This is typically done by reserving a special acknowledgment path in the reverse direction. When the destination accepts a transaction it explicitly ACKs the source; if it discards the transaction it can deliver a negative acknowledgment, informing the source to try again later. Local area networks, such as Ethernet, FDDI, and ATM, take more austere measures and simply drop the transaction whenever there is not space to buffer it. The source relies on time-outs to decide that it may have been dropped and tries again.

Fetch deadlock

The input buffer problem takes on an extra twist in the context of the request/response protocols that are intrinsic to a shared address space and present in message passing implementations. In a reliable network, when a processor attempts to initiate a request transaction the network may refuse to accept it, as a result of contention for the destination and/or contention within the network. In order to keep the network deadlock free, the source is required to continue accepting transactions, even while it cannot initiate its own. However, the incoming transaction may be a request, which will generate a response. The response cannot be initiated because the network is full.

A common solution to this *fetch deadlock* problem is to provide two logically independent communication networks for requests and responses. This may be realized as two physical networks, or separate virtual channels within a single network and separate output and input buffering. While stalled on attempting to send a request it is necessary to continue accepting responses. However, responses can be completed without initiating further transactions. Thus, response transactions will eventually make progress. This implies that incoming requests can eventually be serviced, which implies that stalled requests will eventually make progress.

An alternative solution is to ensure that input buffer space is always available at the destination when a transaction is initiated by limiting the number of outstanding transactions. In a request/response protocol it is straight-forward to limit the number of outstanding requests any processor has outstanding; a counter is maintained and each response decrements the counter allowing a new request to be issued. Standard, blocking reads and writes are realized by simply waiting for the response before completing the current request. Nonetheless, with P processors and a limit of k outstanding requests per processor, it is possible for all kp requests to be directed to the same module. There needs to be space for the kp outstanding requests that might be headed for a single

destination plus space for responses to the requests issued by the destination node. Clearly, the available input buffering ultimately limits the scalability of the system. The request transaction is guaranteed to make progress because the network can always sink transactions into available input buffer space at the destination. The fetch deadlock problem arises when a node attempts to generate a request and its outstanding credit is exhausted. It must service incoming transactions in order to receive its own responses, which enable generation of additional requests. Incoming requests can be serviced because it is guaranteed that the requestor reserved input buffer space for the response. Thus, forward progress is ensured even if the node merely queues and ignores input transactions while attempting to deliver a response.

Finally, we could adopt the approach we followed for split-transaction busses and NAK the transaction if the input buffer is full. Of course, the NAK may be arbitrarily delayed. Here we assume that the network reliably delivers transactions and NAKs, but the destination node may elect to drop them in order to free up input buffer space. Responses never need to be NAKed, because they will be sinked at the destination node, which is the source of the corresponding request and can be assumed to have set aside input buffering for the response. While stalled attempting to initiate a request we need to accept and sink responses and accept and NAK requests. We can assume that input buffer space is available at the destination of the NAK, because it simply uses the space reserved for the intended response. As long as each node provides some input buffer space for requests, we can ensure that eventual some request succeeds and the system does not livelock. Additional precautions are required to minimize the probability of starvation.

7.3.5 Communication architecture design space

In the remainder of this chapter we examine the spectrum of important design points for large-scale distributed memory machines. Recall, our generic large-scale architecture consists of a fairly standard node architecture augmented with hardware communication assist, as suggested by Figure 7-13. The key design issue is the extent to which the information in a network transaction is interpreted directly by the communication assist, without involvement of the node processor. In order to interpret the incoming information, its format must be specified, just as the format of an instruction set must be defined before one can construct an interpreter for it, *i.e.*, a processor. The formatting of the transaction must be performed in part by the source assist, along with address translation, destination routing, and media arbitration. Thus, the processing performed by the source communication assist in generating the network transaction and that at the destination together realize the semantics of the lowest level hardware communication primitives presented to the node architecture. Any additional processing required to realize the desired programming model is performed by the node processor(s), either at user or system level.

Establishing a position on the nature of the processing performed in the two communication assists involved in a network transactions has far reaching implications for the remainder of the design, including how input buffering is performed, how protection is enforced, how many times data is copied within the node, how addresses are translated and so on. The minimal interpretation of the incoming transaction is not to interpret it at all. It is viewed as a raw physical bit stream and is simply deposited in memory or registers. More specific interpretation provides user-level messages, a global virtual address space, or even a global physical address space. In the following sections we examine each of these in turn, and present important machines that embody the respective design points as case studies.
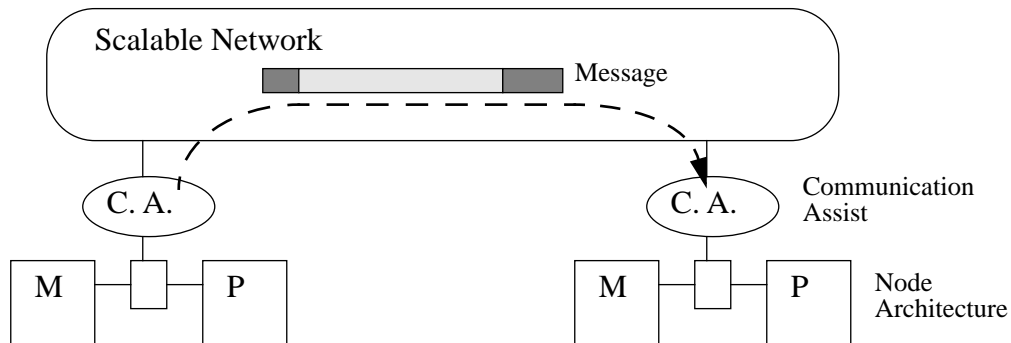
Figure 7-13 Processing of a Network Transaction in the Generic Large-Scale Architecture

A network transaction is a one-way transfer of information from an output buffer at the source to an input buffer at the destination causing some kind of action to occur at the destination, the occurrence of which is not directly visible at the source. The source communication assist formats the transaction and causes it to be routed through the network. The destination communication assist must interpret the transaction and cause the appropriate actions to take place. The nature of this interpretation is a critical design aspect of scalable multiprocessors.

7.4 Physical DMA

This section considers designs where no interpretation is placed on the information within a network transaction. This approach is representative of most early message passing machines, including the nCUBE10, nCUBE/2, the Intel iPSC, iPSC/2, iPSC860, and the Delta, and the Ametek and IBM SP-1. Also, most LAN interfaces follow this approach. The hardware can be very simple and the user communication abstraction can be very general, but typical processing costs are large.

Node to Network Interface

The hardware essentially consists of support for physical DMA (direct memory access), as suggested by Figure 7-14. A DMA device or channel typically has associated with it address and length registers, status (e.g., transmit ready, receive ready), and interrupt enables. Either the device is memory mapped or privileged instructions are provided to access the registers. Addresses are physical[1], so the network transaction is transferred from a contiguous region of memory. To inject the message into the network, requires a trap to the kernel. Typically, the data will be copied into a kernel area so that the envelope, including the route and other information can be constructed. Portions of the envelope, such as the error detection bits, may be generated by the communication assist. The kernel selects the appropriate out-going channel, sets the channel address to the physical address of the message and sets the count. The DMA engine will push the message into the network. When transmission completes the output channel ready flag is set

---

1. One exception to this is the SBus used in Sun workstations and servers, which provide virtual DMA, allowing I/O devices to operate on virtual, rather than physical addresses.

and an interrupt is generated, unless it is masked. The message will work its way through the network to the destination, at which point the DMA at the input channel of the destination node must be started to allow the message to continue moving through the network and into the node. (If there is delay in starting the input channel or if the message collides in the network with another using the same link, typically the message just sits in the network.) Generally, the input channel address register is loaded in advance with the base address where the data is to be deposited. The DMA will transfer words from the network into memory as they arrive. The end-of-message causes the input ready status bit to be set and an interrupt, unless masked. In order to avoid deadlock, the input channels must be activated to receive messages and drain the network, even if there are output messages that need to be sent on busy output channels.

Sending typically requires a trap to the operating system. Privileged software can then provide the source address translation, translate the logical destination node to a physical route, arbitrate for the physical media, and access the physical device. The key property of this approach is that the destination processor initiates a DMA transfer from the network into a region of memory and the next incoming network transaction is blindly deposited in the specified region of memory. The system sets up the in-bound DMA on the destination side. When it opens up an input channel, it cannot determine whether the next message will be a user message or a system message. It will be received blindly into the predefined physical region. Message arrival will typically cause an interrupt, so privileged software can inspect the message and either process it or deliver it to the appropriate user process. System software on the node processor interprets the network transaction and (hopefully) provides a clean abstraction to the user.
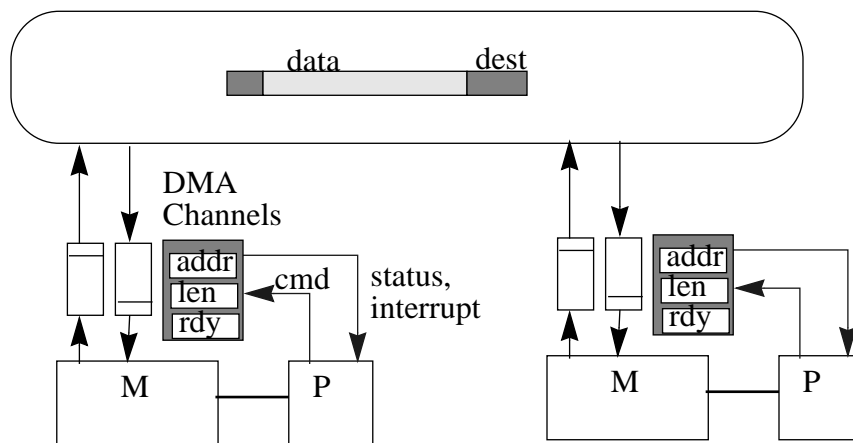
Figure 7-14 Hardware support in the communication assist for blind, physical DMA

The minimal interpretation is blind, physical DMA, which allows the destination communication assist merely to deposit the transaction data into storage, whereupon it will be interpreted by the processor. Since the type of transaction is not determined in advance, kernel buffer storage is used and processing the transaction will usually involve a context switch and one or more copies.

One potential way to reduce the communication overhead is to allow the user level access to the DMA device. If the DMA device is memory mapped, as most are, this is a matter of setting up the user virtual address space to include the necessary region of the I/O space. However, with this approach the protection domain and the level of resource sharing is quite crude. The current user gets the whole machine. If the user misuses the network, there may be no way for the operating system to intervene, other than rebooting the machine. This approach has certainly been

employed in experimental setting, but is not very robust and tends to make the parallel machine into a very expensive personal computer.

Implementing Communication Abstractions

Since the hardware assist in these machines is relatively primitive, the key question becomes how to deliver the newly received network transaction to the user process in a robust, protected fashion. This is where the linkage between programming model and communication primitives occurs. The most common approach is to support the message passing abstraction directly in the kernel. An interrupt is taken on arrival of a network transaction. The process identifier and tag in the network transaction is parsed and a protocol action is taken along the lines specified by Figure 7-10 or Figure 7-12. For example, if a matching receive has been posted the data can be copied directly into the user memory space. If not, the kernel provides buffering or allocates storage in the destination user process to buffer the message until a matching receive is performed. Alternatively, the user process can preallocate communication buffer space and inform the kernel where it wants to receive messages. Some message passing layers allow the receiver to operate directly out of the buffer, rather than receiving the data into its address space.

It is also possible for the kernel software to provide the user-level abstraction of a global virtual address space. In which case, read and write requests are issued either directly through a system call or by trapping on a load or store to a logically remote page. The kernel on the source issues the request and handles the response. The kernel on the destination extracts the user process, command, and destination virtual address from the network transaction and performs the read or write operation, along the lines of Figure 7-8, issuing a response. Of course, the overhead associated with such an implementation of the shared address abstraction is quite large, especially for word at a time operation. Greater efficiency can be gained through bulk data transfers, which make the approach competitive with message passing. There have been many software shared memory systems built along these lines, mostly on clusters of workstations, but the thrust of these efforts is on automatic replication to reduce the amount of communication. They are described in the next chapter.

Other linkages between the kernel and user are possible. For example the kernel could provide the abstraction of a user-level input queue can simply append the message to the appropriate queue, following some well defined policy on queue overflow.

7.4.1 A Case Study: nCUBE/2

A representative example of the physical DMA style of machine is the nCUBE/2. The network interface is organized as illustrated by Figure 7-15, where each of the DMA output channels drives an output port of link and each input DMA channel is associated with an input port. This machine is an example of a *direct network*, in which data is forwarded from its source to its destination through intermediate nodes. The router forwards network transactions from input ports to output ports. The network interface inspects the envelop of messages arriving on each input port and determines whether the message is destined for the local node. If so, the input DMA is activated to drain the message into memory. Otherwise, the router forwards the message to the proper output port.[1] Link-by-link flow control ensures that delivery is reliable. User programs are assigned to contiguous subcubes and the routing is such that the links within the subcube are only used within that subcube, so with space-shared use of the machine, user programs cannot interfere with one another. A peculiarity of the nCUBE/2 is that no count register is associated with
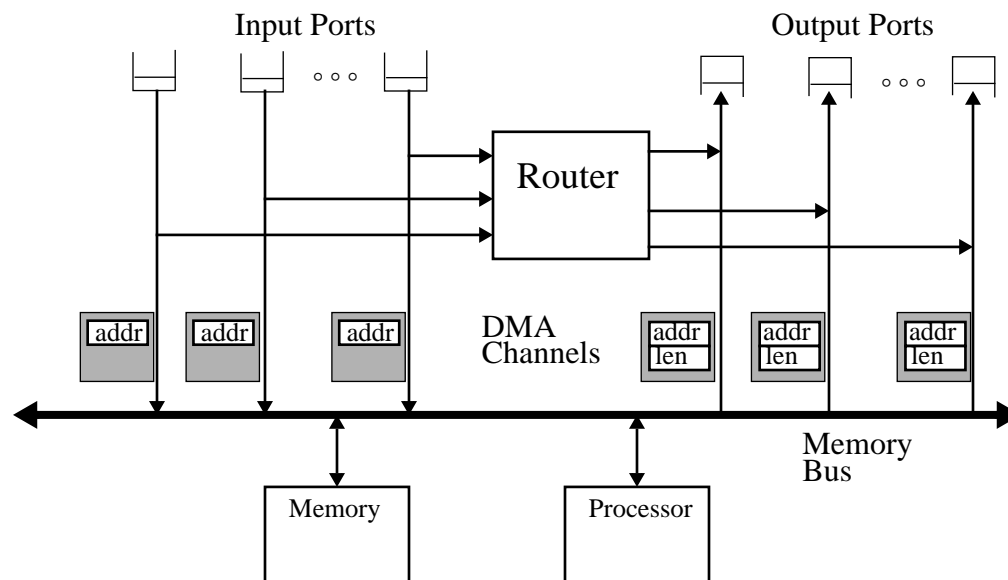
Figure 7-15 Network interface organization of the nCUBE/2

Multiple DMA channels drive network transactions directly from memory into the network or from the network into memory. The inbound channels deposit data into memory at a location determined by the processor, independent of the contents. To avoid copying, the machine allows multiple message segments to be transferred as a single unit through the network. A more typical approach is for the processor to provide the communication assist with a queue of inbound DMA descriptors, each containing the address and length of a memory buffer. When a network transactions arrive, a descriptor is popped from the queue and the data is deposited in the associated memory buffer.

the input channels, so the kernels on the source and destination nodes must ensure that incoming messages never over-run the input buffers in memory.

To assist in interpreting network transactions and delivering the user data into the desired region of memory without copies, it is possible to send a series of message segments as a single, contiguous transfer. At the destination the input DMA will stop at each logical segment. Thus, the destination can take the interrupt and inspect the first segment in order to determine where to direct the remainder, for example by performing the lookup in the receive table. However, this facility is costly, since a start-DMA and interrupt (or busy-wait) is required for each segment.

The best strategy at the kernel level is to keep an input buffer associated with every incoming channel, while output buffers are being injected into the output ports [vEi*92]. Typically each message will contain a header, allowing the kernel to dispatch on the message type and take appropriate action to handle it, such as performing the tag match and copying the message into the user data area. The most efficient and most carefully document communication abstraction on this platform is user-level active messages, which essentially provides the basic network transaction at user level [vEik*92]. It requires that the first word of the user message contain the address of the user routine that will handle the message. The message arrival causes an interrupt, so the

---

1. This routing step is the primary point where the interconnection network topology, an n-cube, is bound into the design of the node. As we will discuss in Chapter 10, the output port is given by the position of the first bit that differs in the local node address and the message destination address.

kernel performs a return-from-interrupt to the message handler, with the interrupted user address on the stack. With this approach, a message can be delivered into the network in 13 μs (16 instructions, including 18 memory references, costing 160 cycles) and extracted from the network in 15 μs (18 instructions, including 26 memory references, costing 300 cycles). Comparing this with the 150 μs start-up of the vendor "vertex" message passing library reflects the gap between the hardware primitives and the user-level operations in the message passing programming model. The vertex message passing layer uses an optimistic one-way protocol, but matching and buffer management is required.

Typical LAN Interfaces

The simple DMA controllers of the nCUBE/2 are typical of parallel machines and qualitatively different from what is typically found in DMA controllers for peripheral devices and local area networks. Notice that each DMA channel is capable of a single, contiguous transfer. A short instruction sequence sets the channel address and channel limit for the next input or output operation. Traditional DMA controllers provide the ability to chain together a large number of transfers. To initiate an output DMA, a DMA descriptor is chained onto the output DMA queue. The peripheral controller polls this queue, issuing DMA operations and informing the processor as they complete.

Most LAN controllers, including Ethernet LANCE, Motorola DMAC, Sun ATM adapters and many others, provide a queue of transmit descriptors and a queue of receive descriptors. (There is also a free list of each kind of descriptor. Typically, the queue and its free list are combined into a single ring.) The kernel builds the output message in memory and sets up a transmit descriptor with its address and length, as well as some control information. In some controllers a single message can be described by a sequence of descriptors, so the controller can gather the envelope and the data from memory. Typically, the controller has a single port into the network, so it pushes the message onto the wire. For Ethernets and rings, each of the controllers inspect the message as it comes by, so a destination address is specified on the transaction, rather than a route.

The inbound side is more interesting. Each receive descriptor has a destination buffer address. When a message arrives, a buffer descriptor is popped off the queue and a DMA transfer is initiated to load the message data into memory. If no receive descriptor is available, the message is dropped and higher level protocols must retry (just as if the message was garbled in transit). Most devices have configurable interrupt logic, so an interrupt can be generated on every arrival, after so many bytes, or after a message has waited for too long. The operating system driver manages these input and output queues. The number of instructions required to set up even a small transfer is quite large with such devices, partly because of the formatting of the descriptors and the handshaking with the controller.

7.5 User-level Access

The most basic level of hardware interpretation of the incoming network transaction distinguishes user messages from system messages and delivers user messages to the user program without operating system intervention. Each network transaction carries a user/system flag which is examined by the communication assist as the message arrives. In addition, it should be possible to inject a user message into the network at user level; the communication assist automatically

inserts the user flag as it generates the transaction. In effect, this design point provides a *user-level network port*, which can be written and read without system intervention.

Node to Network Interface

A typical organization for a parallel machine supporting user-level network access is shown in Figure 7-16. A region of the address space is mapped to the network input and output ports, as well as the status register, as indicated in Figure 7-17. The processor can generate a network transaction by writing the destination node number and the data into the output port. The communication assist performs protection check, translates the logical destination node number into a physical address or route, and arbitrates for the medium. It also inserts the message type and any error checking information. Upon arrival, a system message will cause an interrupt so the system can extract it from the network, whereas a user message can sit in the input queue until the user process reads it from the network, popping the queue. If the network backs up, attempts to write messages into the network will fail and the user process will need to continue to extract messages from the network to make forward progress. Since current microprocessors do not support user-level interrupts, an interrupting user message is treated by the NI as a system message and the system rapidly transfers control to a user-level handler.
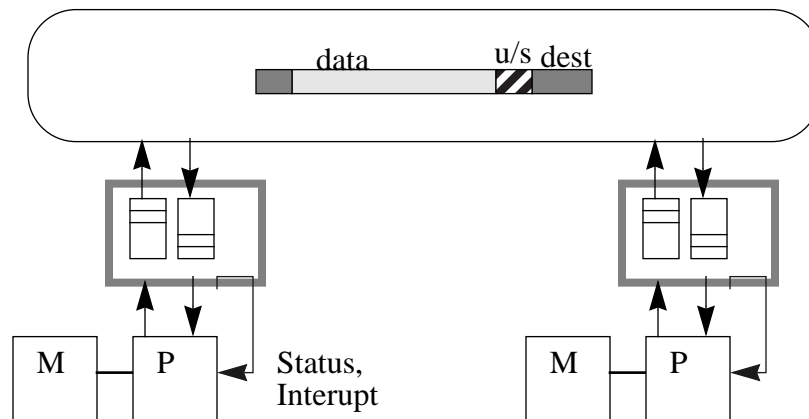
Figure 7-16 Hardware support in the communication assist for user-level network ports.

The network transaction is distinguished as either system or user. The communication assist provides network input and output fifos accessible to the user or system. It marks user messages as they are sent and checks the transaction type as they are received. User messages may be retained in the user input fifo until extracted by the user application. System transactions cause an interrupt, so that they may be handled in a priviledged manner by the system. In the absence of user-level interupt support, interrupting user tranactions are treated as special system transactions.

One implication of this design point is that the communication primitives allow a portion of the process state to be in the network, having left the source but not arrived at the destination. Thus, if the collection of user processes forming a parallel program is time-sliced out, the collection of in-flight messages for the program need to be swapped as well. They will be reinserted into the network or the destination input queues when the program is resumed.
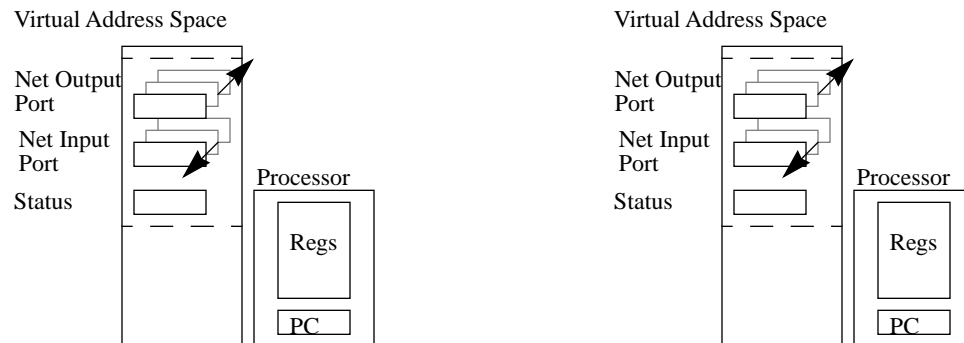
Figure 7-17 Typical User-level Architecture with network ports

In addition to the storage presented by the instruction set architecture and memory space, a region of the user's virtual address space provides access to the network output port, input port, and status register. Network transactions are initiated and received by writing and reading the ports, plus checking the status register.

7.5.1 Case Study: Thinking Machines CM-5

The first commercial machine to seriously support a user-level network was the Thinking Machines Corp. CM-5, introduced in late 1991. The communication assist is contained in a network interface chip (NI) attached at the memory bus as if it were an additional memory controller, as illustrated in Figure 7-12. The NI provides an input and output FIFO for each of two data networks and a "control network", which is specialized for global operations, such as barrier, broadcast, reduce and scan. The functionality of the communication assist is made available to the processor by mapping the network input and output ports, as well as the status registers, into the address space, as shown in Figure 7-17. The kernel can access all of the FIFOs and registers, whereas a user process can only access the user FIFO and status. In either case, communication operations are initiated and completed by reading and writing the communication assist registers using conventional loads and stores. In addition, the communication assist can raise an interrupt. In the CM5, each network transaction contains a small tag and the communication assist maintains a small table to indicate which tags should raise an interrupt. All system tags raise an interrupt.

In the CM-5, it is possible to write a five word message into the network in 1.5 μs (50 cycles) and read one out in 1.6 μs. In addition, the latency across an unloaded network varies from 3 μs for neighboring nodes to 5 μs across a thousand node machine. An interrupt vectored to user level costs roughly 10 μs. The user-level handler may process several messages, if they arrive in rapid succession. The time to transfer a message into or out of the network interface is dominated by the time spent on the memory bus, since these operations are performed as uncached writes and reads. If the message data starts out in registers, it can be written to the network interface as a sequence of buffered stores. However, this must be followed by a load to check if the message was accepted, at which point the write latency is experienced as the write buffer is retired to the NI.

If the message data originates in memory and is to be deposited in memory, rather than registers, it is interesting to evaluate whether DMA should be used to transfer data to and from the NI. The critical resource is the memory bus. When using conventional memory operations to access the

user-level network port, each word of message data is first loaded into a register and then stored into the NI or memory. If the data is uncachable, each data word in the network transaction involves four bus transactions. If the memory data is cachable, the transfers between the processor and memory are performed as cache block transfers or are avoided if the data is in cache. However, the NI stores and loads remain. With DMA transfers, the data is moved only once across the memory bus on each end of the network transaction, using burst mode transfers. However, the DMA descriptor must still be written to the NI. The performance advantages of DMA can be lost altogether if the message data cannot be in cachable regions of memory, so the DMA transfer performed by the NI must be coherent with respect to the processor cache. Thus, it is critical that the node memory architecture support coherent caching. On the receiving side, the DMA must be initiated by the NI based on information in the network transaction and state internal to the NI; otherwise we are again faced with the problems associated with blind physical DMA. This leads us to place additional interpretation on the network transaction, in order to have the communication assist extract address fields. We consider this approach further in Section 7.6 below.

The two data networks in the CM-5 provide a simple solution to the fetch deadlock problem; one network can be used for requests and one for responses[Lei*92]. When blocking on a request, the node continues to accept incoming replies and requests, which may generate out-going replies. When blocked on sending a reply, only incoming replies are accepted from the network. Eventually the reply will succeed, allowing the request to proceed. Alternatively, buffering can be provided at each node, with some additional end-to-end flow control to ensure that the buffers do not overflow. Should a user program be interrupted when it is part way through popping a message from the input queue, the system will extract the remainder of the message and pushes it back into the front of the input queue before resuming the program.

7.5.2 User Level Handlers

Several experimental architectures have investigated a tighter integration of the user-level network port with the processor, including the Manchester Dataflow Machine[Gur*85], Sigma-1[Shi*84], iWARP[Bor90], Monsoon[PaCu90], EM-4[Sak*91], and J-Machine[Dal93]. The key difference is that the network input and output ports are processor registers, as suggested by Figure 7-18, rather than special regions of memory. This substantially changes the engineering of the node, since the communication assist is essentially a processor function unit. The latency of each of the operations is reduced substantially, since data is moved in and out of the network with register-to-register instructions. The bandwidth demands on the memory bus are reduced and the design of the communication support is divorced from the design of the memory system. However, the processor is involved in every network transaction. Large data transfers consume processor cycles and are likely to pollute the processor cache.

Interestingly, the experimental machines have arrived at a similar design point from vastly different approaches. The iWARP machine[Bor*90], developed jointly by CMU and Intel, binds two registers in the main register file to the head of the network input and output ports. The processor may access the message on a word-by-word basis as it streams in from the network. Alternatively, a message can be spooled into memory by a DMA controller. The processor specifies which message it desires to access via the port registers by specifying the message tag, much as in a traditional receive call. Other messages are spooled into memory by the DMA controller using an input buffer queue to specify the destination address. The extra hardware mechanism to direct one in-coming and one out-going message through the register file was motivated by sys-
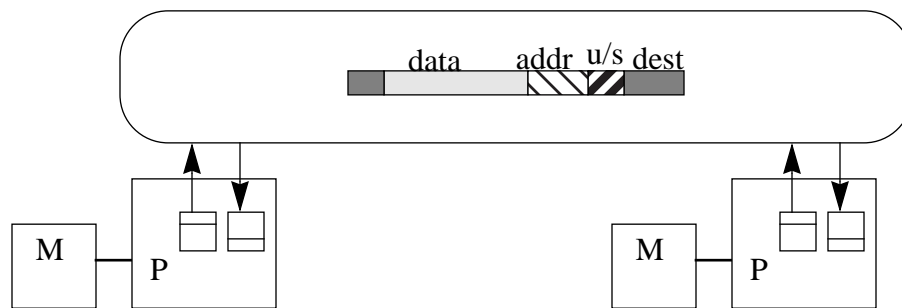
Figure 7-18 Hardware support in the communication assist for user-level handlers.

The basic level of support required for user-level handlers is that the communication assist can determine that the network transaction is destined for a user processes and make it directly available to that process. This either means that each process has a logical set of FIFOs or a single set FIFOs is timeshared among user processes.

tolic algorithms where a stream of data is pumped through processors in a highly regular pipeline, doing a small amount of computation as the stream flows through. By interpreting the tag, or virtual channel in iWARP terms, in the network interface, the memory-based message flows are not hindered by the register-based message flow. By contrast, in a CM-5 style of design all messages are interleaved through a single input buffer.

The *T machine[NPA93], proposed by MIT and Motorola, offered a more general-purpose architecture for user-level message handling. It extended the Motorola 88110 RISC microprocessor to include a network function unit containing a set of registers much like the floating-point unit. A multiword outgoing message is composed in a set of output registers and a special instruction causes the network function unit to send it out. There are several of these output register sets forming a queue and the send advances the queue, exposing the next available set to the user. Function unit status bits indicate whether an output set is available; these can be used directly in branch instructions. There are also several input message register sets, so when a message arrives it is loaded into an input register set and a status bit is set or an interrupt is generated. Additional hardware support was provided to allow the processor to dispatch rapidly to the address specified by the first word of the message.

The *T design drew heavily on previous efforts supporting message driven execution and dataflow architectures, especially the J-Machine, Monsoon and EM-4. These earlier designs employed rather unusual processor architectures, so the communication assist is not clearly articulated. The J-machine provides two execution contexts, each with a program counter and small register set. The "system" execution context has priority over the "user" context. The instruction set includes a segmented memory model, with one segment being a special on-chip message input queue. There is also a message output port for each context. The first word of the network transaction is specified as being the address of the handler for the message. Whenever the user context is idle and a message is present in the input queue, the head of the message is automatically loaded into the program counter and an address register is set up to reference the rest of the message. The handler must extract the message from the input buffer before suspending or competing. Arrival of a system level message preempts the user context and initiates a system handler.

In Monsoon, a network transaction is of fixed format and specified to contain a handler address, data frame address, and a 64-bit data value. The processor supports a large queue of such small messages. The basic instruction scheduling mechanism and the message handling mechanism are deeply integrated. In each instruction fetch cycle, a message is popped from the queue and the instruction specified by the first word of the message is executed. Instructions are in a 1+x address format and specify an offset relative to the frame address where a second operand is located. Each frame location contains *presence bits*, so if the location is empty the data word of the message is stored in the specified location (like a store accumulator instruction). If the location is not empty, its value is fetched, an operation is performed on the two operands, and one or more messages carrying the result are generated, either for the local queue or a queue across the network. In earlier, more traditional dataflow machines, the network transaction carried an instruction address and a tag, which is used in an associative match to locate the second operand, rather simple frame relative addressing. Later hybrid machines [NiAr89,GrHo90,Cul*91,Sak91] execute a sequence of instructions for each message dequeue-and-match operation.

7.6 Dedicated Message Processing

A third important design style for large-scale distributed-memory machines seeks to allow sophisticated processing of the network transaction using dedicated hardware resources without binding the interpretation in the hardware design. The interpretation is performed by software on a dedicated communications processor (or message processor) which operates directly on the network interface. With this capability, it is natural to consider off-loading the protocol processing associated with the message passing abstraction to the communications processor. It can perform the buffering, matching, copying and acknowledgment operations. It is also reasonable to support a global address space where the communication processor performs the remote read operation on behalf of the requesting node. The communications processors can cooperate to provide a general capability to move data from one region of the global address space to another. The communication processor can provide synchronization operations and even combinations of data movement and synchronization, such as write data and set a flag or enqueue data. Below we look at the basic organizational properties of machines of this class to understand the key design issues. We will look in detail at two machines as case studies, the Intel Paragon and the Meiko CS-2.

A generic organization for this style of design is shown in Figure 7-19, where the compute processor (P) and communication processor (mP) are symmetric and both reside on the memory bus. This essentially starts with a bus-based SMP of Chapter 4 as the node and extends it with a primitive network interface similar to that described in the previous two sections. One of the processors in the SMP node is specialized in software to function as a dedicated message processor. An alternative organization is to have the communication processor embedded into the network interface, as shown in Figure 7-20. These two organizations have different latency, bandwidth, and cost trade-offs, which we examine a little later. Conceptually, they are very similar. The communication processor typically executes at system privilege level, relieving the machine designer from addressing the issues associated with a user-level network interface discussed above. The two processors communicate via shared memory, which typically takes the form of a command queue and response area, so the change in privilege level comes essentially for free as part of the hand-off. Since the design assumes there is a system-level processor responsible for managing network transactions, these designs generally allow word-by-word access to the NI FIFOs, as well as DMA, so the communication processor can inspect portions of the message and decide
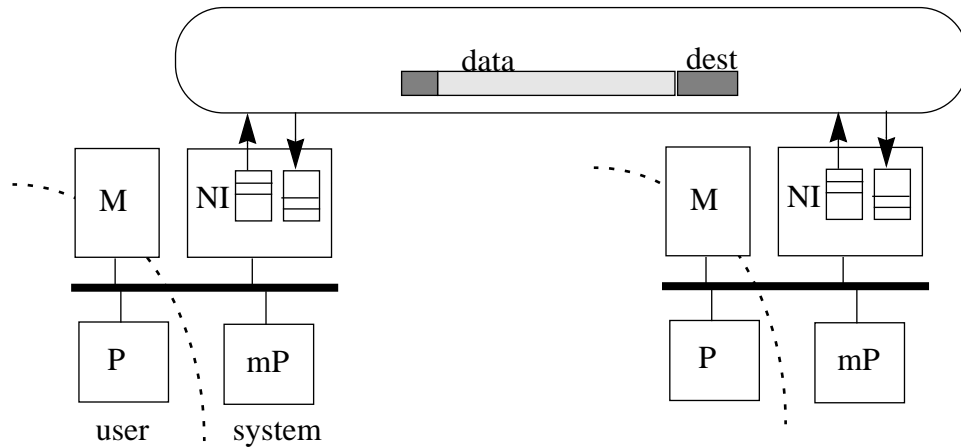
Figure 7-19 Machine organization for dedicated message processing with a symmetric processor

Each node has a processor, symetric with the main processor on a shared memory bus, that is dedicated to initiating and handling network transactions. Being dedicated, it can always run in system mode, so transferring data through memory implicitly crosses the protection boundary. The message processor can provide any additional protection checks of the contents of the transactions.

what actions to take. The communication processor can poll the network and the command queues to move the communication process along.
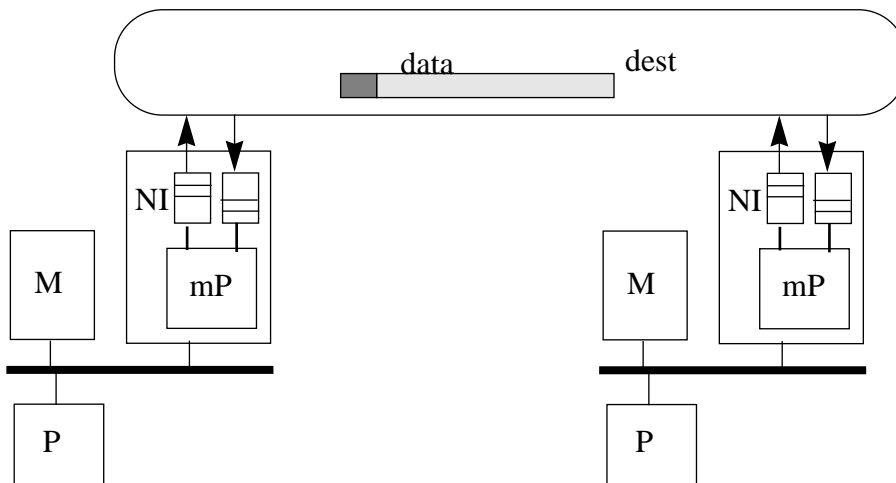
Figure 7-20 Machine organization for dedicated message processing with an embedded processor

The communication assist consists of a dedicated, programmable message processor embedded in the network interface. It has a direct path to the network that does not utilize the memory bus shared by the main processor.

The message processor provides the computation processor with a very clean abstraction of the network interface. All the details of the physical network operation are hidden, such as the hardware input/output buffers, the status registers, and the representation of routes. A message can be sent by simply writing it, or a pointer to it, into shared memory. Control information is exchanged between the processors using familiar shared-memory synchronization primitives, such as flags and locks. Incoming messages can be delivered directly into memory by the mes-

sage processor, along with notification to the compute processor via shared variables. With a well-designed user-level abstraction, the data can be deposited directly into the user address space. A simple low-level abstraction provides each user process in a parallel program with a logical input queue and output queue. In this case the flow of information in a network transaction is as shown in Figure 7-21.
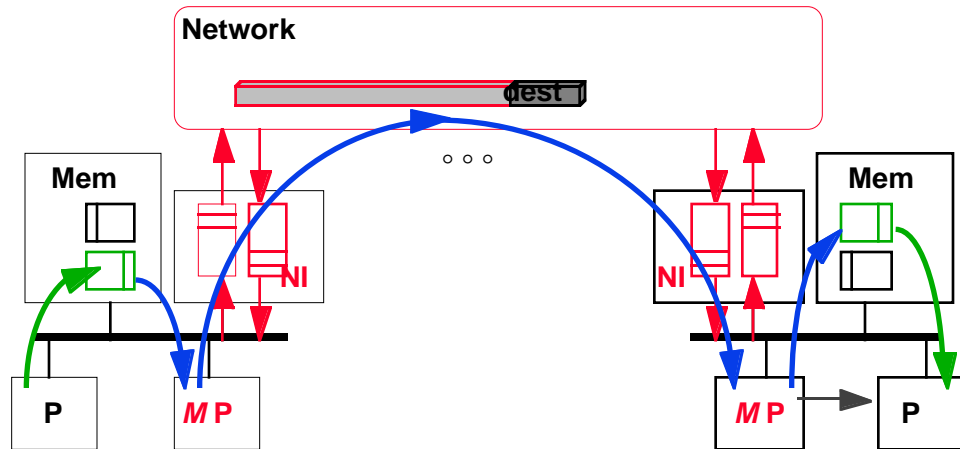
Figure 7-21 Flow of a Network Transaction with a symmetric message processor

Each network transaction flows through memory, or at least across the memory bus in a cache-to-cache transfer between the main processor and memory processor. It crosses the memory bus again between the message processor and network interface.

These benefits are not without costs. Since communication between the compute processor and message processor is via shared memory within the node, communication performance is strongly influenced by the efficiency of the cache coherency protocol. A review of Chapter 4 reveals that these protocols are primarily designed to avoid unnecessary communication when two processors are operating on mostly distinct portions of a shared data structure. The shared communication queues are a very different situation. The producer writes an entry and sets a flag. The consumer must see the flag update, read the data, and clear the flag. Eventually, the producer will see the cleared flag and rewrite the entry. All the data must be moved from the producer to the consumer with minimal latency. We will see below that inefficiencies in traditional coherency protocols make this latency significant. For example, before the producer can write a new entry, the copy of the old one in the consumer's cache must be invalidated. One might imagine that an update protocol, or even uncached writes might avoid this situation, but then a bus transaction will occur for every word, rather than every cache block.

A second problem is that the function performed by the message processor is "concurrency intensive". It handles requests from the compute processor, messages arriving from the network and messages going out and into the network all at once. By folding all these events into a single sequential dispatch loop, they can only be handled one at a time. This can seriously impair the bandwidth capability of the hardware.

Finally, the ability of the message processor to deliver messages directly into memory does not completely eliminate the possibility of fetch deadlock at the user level, although it can ensure that

the physical resources are not stalled when an application deadlocks. A user application may need to provide some additional level of flow control.

7.6.1 Case Study: Intel Paragon

To make these general issues concrete, in this section we examine how they arise in an important machine of this ilk, the Intel Paragon, first shipped in 1992. Each node is a shared memory multiprocessor with two or more 50MHz i860XP processors, a Network Interface Chip (NIC) and 16 or 32 MB of memory, connected by a 64-bit, 400 MB/sec, cache-coherent memory bus, as shown in Figure 7-22. In addition, two DMA engines (one for sending and the other for receiving) are provided to burst data between memory and the network. DMA transfers operate within the cache coherency protocol and are throttled by the network interface before buffers are over or under-run. One of the processors is designated as a message processor to handle network transactions and message passing protocols while the other is used as a compute processor for general computing. 'I/O nodes' are formed by adding I/O daughter cards for SCSI, Ethernet, and HiPPI connections.
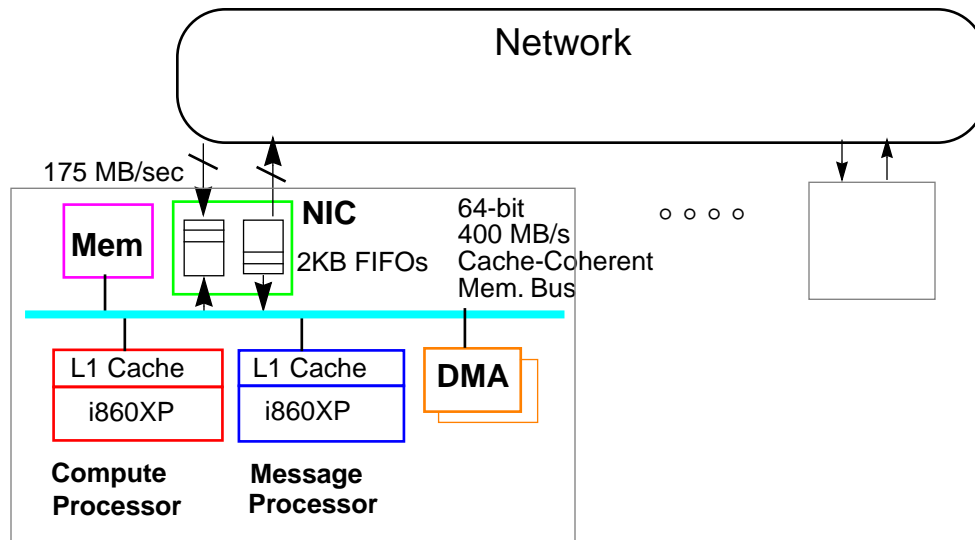
Figure 7-22 Intel Paragon machine organization

Each node of the machine includes a dedicated message processor, identical to the compute processor on a cache coherent memory bus, which has a simple, system-level interface to a fast, reliable network, and two cache-coherent DMA engines that repects the flow control of the NIC

The i860XP processor is a 50MHz RISC processor that supports dual-instruction issue (one integer and one floating-point), dual-operation instructions (e.g. floating-point multiply-and-add) and graphics operations. It is rated at 42 MIPS for integer operations and has a peak performance of 75 MFLOPS for double-precision operations. The paging unit of the i860XP implements protected, paged, virtual memory using two four-way set-associative translation look-aside buffers (TLB). One TLB supports 4KB pages using two-level page tables and has 64 entries; the other supports 4MB pages using one-level page tables and has 16 entries. Each i860XP has a 16KB instruction cache and a 16KB data cache, which are 4-way set-associative with 32-byte cache blocks [i860]. The i860XP uses write-back caching normally, but it can also be configured to use write-through and write-once polices under software or hardware control. The write buffers can

hold two successive stores to prevent stalling on write misses. The cache controllers of the i860XP implement a variant of the MESI (Modified, Exclusive, Shared, Invalid) cache consistency protocol discussed in Chapter 4. The external bus interface also supports a 3-stage address pipeline (i.e. 3 outstanding bus cycles) and burst mode with transfer length of 2 or 4 at 400 MB/sec.

The Network Interface Chip (NIC) connects the 64-bit synchronous memory bus to 16-bit asynchronous (self-timed) network links. A 2KB transmit FIFO (tx) and a 2KB receive FIFO (rx) are used to provide rate matching between the node and a full-duplex 175 MB/sec network link. The head of the rx FIFO and the tail of the tx FIFO are accessible to the node as a memory mapped NIC I/O register. In addition, a status register contains flags that are set when the FIFO is full, empty, almost full, or almost empty and when an end-of-packet marker is present. The NIC can optionally generate an interrupt when each flag is set. Reads and writes to the NIC FIFOs are uncached and must be done one double-word (64 bits) at a time. The first word of a message must contain the route (X-Y displacements in a 2D mesh), but the hardware does not impose any other restriction on the message format. In particular, it does not distinguish between system and user messages. In addition, the NIC also performs parity and CRC checks to maintain end-to-end data integrity.

Two DMA engines, one for sending and the other for receiving, can transfer a contiguous block of data between main memory and the Network Interface Chip (NIC) at 400MB/sec. The memory region is specified as a physical address, aligned on a 32-byte boundary, with a length between 64 bytes and 16KB (one DRAM page) in multiples of 32 bytes (a cache block). During DMA transfer, the DMA engine snoops on the processor caches to ensure consistency. Hardware flow control prevents the DMA from overflowing or underflowing the NIC FIFO's. If the output buffer is full, the send DMA will pause and free the bus. Similarly, the receive DMA pauses when the input buffer is empty. The bus arbitrator gives priority to the DMA engine over the processors. A DMA transfer is started by storing an address-length pair to a memory mapped DMA register using the `stio` instruction. Upon completion, the DMA engine sets a flag in a status register and optionally generates an interrupt.

With this hardware configuration a small message of just less than two cache blocks (seven words) can be transferred from registers in one compute processor to registers in another compute processor waiting on the transfer in just over 10 μs (500 cycles). This time breaks down almost equally between the three processor-to-processor transfers: compute processor to message processor across the bus, message processor to message processor across the network, and message processor to compute processor across the bus on the remote node. It may seem surprising that transfers between two processors on a cache-coherent memory bus would have the same latency as transfers between two message processors through the network, especially since the transfers between the processor and the network interface involves a transfer across the same bus.

Let's look at this situation in a little more detail. An i860 processor can write a cache block from registers using two quad-word store instructions. Suppose that part of the block is used as a full/empty flag. In the typical case, the last operation on the block was a store by the consumer to clear the flag; with the Paragon MESI protocol this writes through to memory, invalidates the producer block, and leaves the consumer in the exclusive state. The producer's load on the flag which finds the flag clear misses in the producer cache, reads the block from memory, and downgrades the consumer block to shared state. The first store writes through and invalidates the consumer block, but it leaves the producer block in shared state since sharing was detected when the write was performed. The second store also writes-through, but since there is no sharer it leaves

the producer in the exclusive state. The consumer eventually reads the flag, misses, and brings in the entire line. Thus, four bus transactions are required for a single cache block transfer. (By having an additional flag that allows the producer to check that several blocks are empty, this can be reduced to three bus transactions. This is left as an exercise.) Data is written to the network interface as a sequence of uncached double-word stores. These are all pipelined through the write-buffer, although they do involve multiple bus transfers. Before writing the data, the message processor needs to check that there is room in the output buffer to hold it. Rather than pay the cost of an uncached read, it checks a bit in the processor status word corresponding to a masked 'output buffer empty' interrupt. On the receiving end, the communication processor reads a similar 'input non-empty' status bit and then reads the message data as a series of uncached loads. The actual NI to NI transfer takes only about 250ns plus 40ns per hop in an unloaded network.

For a bulk memory-to-memory transfer, there is additional work for the communication processor to start the send-DMA from the user source region. This requires about $2\,\mu s$ (100 cycles). The DMA transfer bursts at 400 MB/s into the network output buffer of 2048 bytes. When the output buffer is full, the DMA engine backs off until a number of cache blocks are drained into the network. On the receiving end, the communication processor detects the presence of an incoming message, reads the first few words containing the destination memory address, and starts the receive-DMA to drain the remainder of the message into memory. For a large transfer, the send DMA engine will be still moving data out of memory, the receive DMA engine will be moving data into memory, and a portion of the transfer will occupy the buffers and network links in between. At this point, the message moves forward at the 175 MB/s of the network links. The send and receive DMA engines periodically kick in and move data to or from network buffers at 400 MB/s bursts.

A review of the requirements on the communication processor shows that it is responding to a large number of independent events on which it must take action. These include the current user program writing a message into the shared queue, the kernel on the compute processor writing a message into a similar "system queue", the network delivering a message into the NI input buffer, the NI output buffer going empty as a result of the network accepting a message, the send DMA engine completing, and the receive DMA engine completing. The bandwidth of the communication processor is determined by the time it takes to detect and dispatch on these various events. While handling any one of the events all the others are effectively locked out. Additional hardware, such as the DMA engines, is introduced to minimize the work in handling any particular event and allow data to flow from the source storage area (registers or memory) to the destination storage area in a fully pipelined fashion. However, the communication rate (messages per second) is still limited by the sequential dispatch loop in the communication processor. Also, the software on the computation processor which keeps data flowing, avoids deadlock, and avoids starving the network is rather tricky. Logically it involves a number of independent cooperating threads, but these are folded into a single sequential dispatch loop which keeps track of the state of each of the partial operations. This concurrency problem is addressed by our next case study.

The basic architecture of the Paragon is employed in the ASCI Red machine, which is the first machine to sustain a teraFLOPS (one trillion floating-point operations per second). When fully configured, this machine will contain 4500 nodes with dual 200 MHz Pentium Pro processors and 64 MB of memory. It uses an upgraded version of the Paragon network with 400 MB/s links, still in a grid topology. The machine will be spread over 85 cabinets, occupying about 1600 sq. ft and drawing 800 KW of power. Forty of the nodes will provide I/O access to large RAID storage systems, an additional 32 nodes will provide operating system services to a lightweight kernel operating on the individual nodes, and 16 nodes will provide "hot" spares.

Many cluster designs employ SMP nodes as the basic building block, with a scalable high performance LAN or SAN. This approach admits the option of dedicating a processor to message processing or of having that responsibility taken on by processors as demanded by message traffic. One key difference is that networks such as that used in the Paragon will back up and stop all communication progress, including system messages, unless the inbound transactions are serviced by the nodes. Thus, dedicating a processor to message handling provides a more robust design. (In special cases, such as attempting to set the record on the Linpack benchmark, even the Paragon and ASCI red are run with both processors doing user computation.) Clusters usually rely on other mechanisms to keep communication flowing, such as dedicating processing within the network interface card, as we discuss below.

7.6.2   Case Study: Meiko CS-2

The Meiko CS-2 provides a representative concrete design with an asymmetric message processor, that is closely integrated with the network interface and has a dedicated path to the network. The node architecture is essentially that of a SparcStation 10, with two standard superscalar Sparc modules on the MBUS, each with L1 cache on-chip and L2 cache on the module. Ethernet, SBUS, and SCSI connections are also accessible over the MBUS through a bus adapter to provide I/O. (A high performance variant of the node architecture includes two Fujistsu μVP vector units sharing a three ported memory system. The third port is the MBUS, which hosts the two compute processors and the communications module, as in the basic node.) The communications module functions as either another processor module or a memory module on the MBUS, depending on its operation. The network links provide 50 MB/s bandwidth in each direction. This machine takes a unique position on how the network transaction is interpreted and on how concurrency is supported in communication processing.

A network transaction on the Meiko CS-2 is a code-sequence transferred across the network and executed directly by the remote communications processor. The network is *circuit switched*, so a channel is established and held open for the duration of the network transaction execution. The channel closes with an ACK if the channel was established and the transaction executed to completion successfully. A NAK is returned if connection is not established, there is a CRC error, the remote execution times out, or a conditional operation fails. The control flow for network transactions is straight-line code with conditional abort, but no branching. A typical cause of time-out is a page-fault at the remote end. The sorts of operations that can be included in a network transactions include: read, write, or read-modify-write of remote memory, setting events, simple tests, DMA transfers, and simple reply transactions. Thus, the format of the information in a network transaction is fairly extensive. It consists of a context identifier, a start symbol, a sequence of operations in a concrete format, and an end symbol. A transaction is between 40 and 320 bytes long. We will return to the operations supported by network transactions in more detail after looking at the machine organization.

Based on our discussion above, it makes sense to consider decomposing the communication processor into several, independent processors, as indicated by Figure 7-23. A command processor (P_{cmd}) waits for communication commands to be issued on behalf of the user or system and carries them out. Since it resides as a device on the memory bus, it can respond directly to reads and writes of addresses for which it is responsible, rather than polling a shared memory location as would a conventional processor. It carries out its work by pushing route information and data into the output processor (P_{out}) or by moving data from memory to the output processor. (Since the output processor acts on the bus as a memory device, each store to its internal buffers can trigger

it to take action.) It may require assistance of a device responsible for virtual-to-physical (V->P) address translation. It also provides whatever protection checks are required for user-to-user communication. The output processor monitors status of the network output FIFO and delivers network transactions into the network. An input processor (P_{in}) waits for arrival of a network transaction and executes it. This may involve delivering data into memory, posting a command to an event processor (P_{event}) to signal completion of the transaction or posting a reply operation to a reply processor (P_{reply})which operates very much like the output processor.
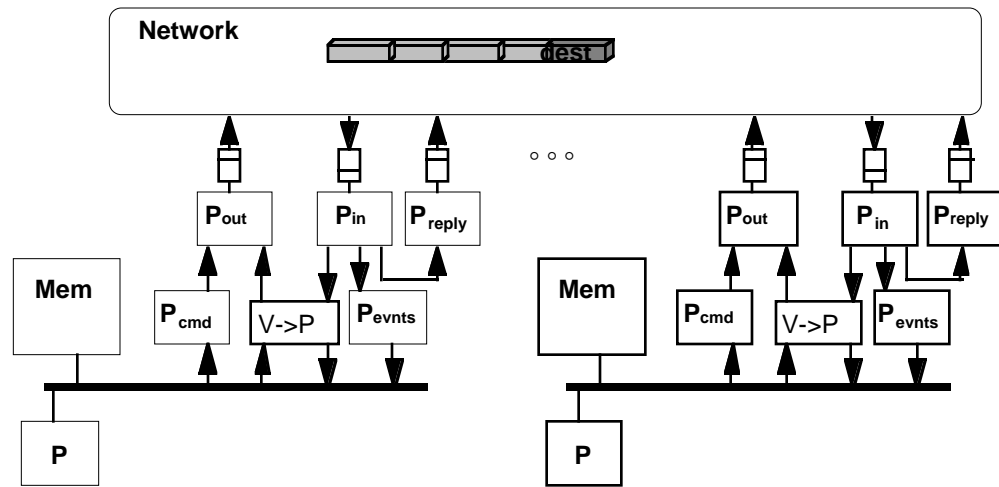
Figure 7-23 Meiko CS-2 Conceptual Structure with multiple specialized communication processors

Each of the individual aspects of generating and processing network transactions is associated in independently operating hardware function units.

The Meiko CS-2 provides essentially these independent functions, although they operate as time-multiplexed threads on a single microprogrammed processor, called the *elan* [HoMc93]. This makes the communication between the logical processors very simple and provides a clean conceptual structure, but it does not actually keep all the information flows progressing smoothly. The function organization of Elan is depicted in Figure 7-24. A command is issued by the compute processor to the command processor via an exchange instruction, which swaps the value of a register and a memory location. The memory location is mapped to the head of the command processor input queue. The value returned to the processor indicates whether the enqueue command was successful or whether the queue was already full. The value given to the command processor contains a command type and a virtual address. The command processor basically supports three commands: start DMA, in which case the address points to a DMA descriptor, set event, in which case the address refers to a simple event data structure, or start thread, in which case the address specifies the first instruction in the thread. The DMA processor reads data from memory and generates a sequence of network transactions to cause data to be stored into the remote node. The command processor also performs event operations, which involve updating a small event data structure and possibly raising an interrupt for the main processor in order to wake a sleeping thread. The start-thread command is conveyed to a simple RISC thread processor which executes an arbitrary code sequence to construct and issue network transactions. Network transactions are interpreted by the input processor, which may cause threads to execute, DMA to start, replies to be issued, or events to be set. The reply is simply a set-event operation with an optional write of three words of data.
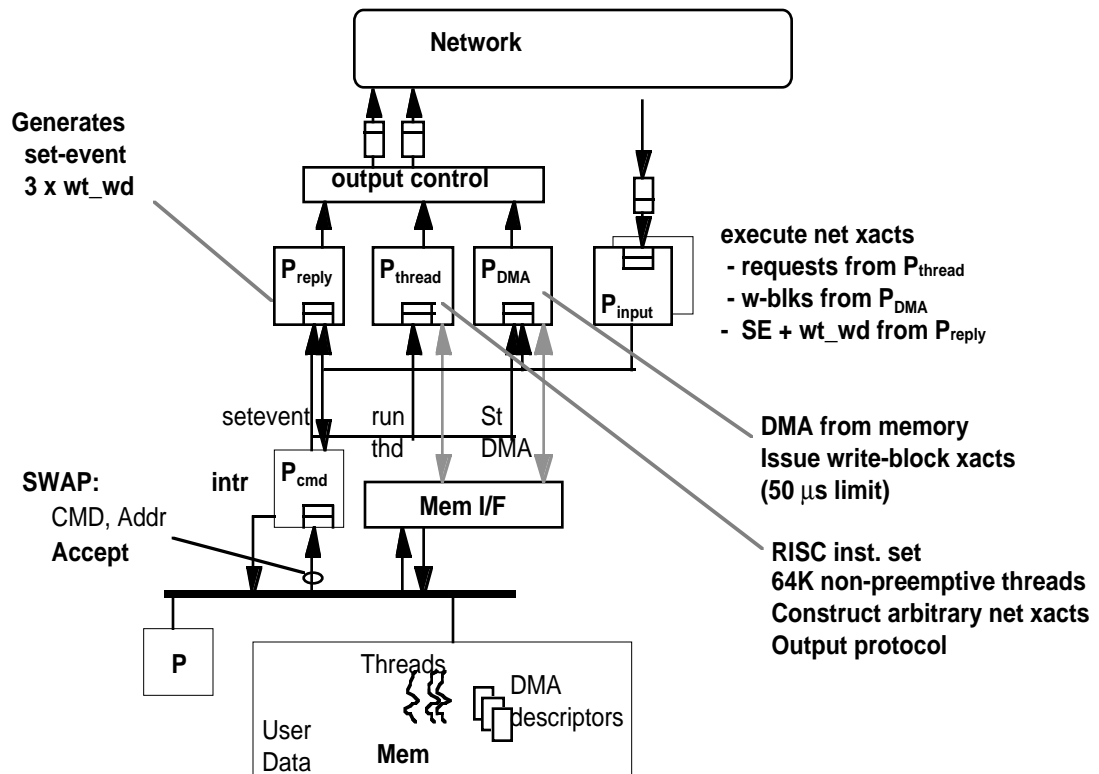
Figure 7-24 Meiko CS-2 machine organization

To make the machine operation more concrete, let us consider a few simple operations. Suppose a user process wants to write data into the address space of another process of the same parallel program. Protection is provided by a capability for a communication context which both processes share. The source compute processor builds a DMA descriptor and issues a start DMA command. Its DMA processor reads the descriptor and transfers the data as a sequence of block, each involves loading up to 32 bytes from memory and forming a write_block network transaction. The input processor on the remote node will receive and execute a series of write_block transactions, each containing a user virtual memory address and the data to be written at that address. Reading a block of data from a remote address space is somewhat more involved. A thread is started on the local communication processor which issues a start-DMA transaction. The input processor on the remote node passes the start-DMA and its descriptor to the DMA processor on the remote node, which reads data from memory and returns it as a sequence of write_block transactions. To detect completion, these can be augmented with set-event operations.

In order to support direct user-to-user transfers, the communication processor on the Meiko CS-2 contains its own page table. The operating system on the main processor keeps this consistent with the normal page tables. If the communication processor experiences a page fault, an interrupt is generated at the processor so that the operating system there can fault in the page and update the page tables.

The major shortcoming with this design is that the thread processor is quite slow and is non-pre-emptively scheduled. This makes it very difficult to off-load any but the most trivial processing to the thread processor. Also, the set of operations provided by network transactions are not power-ful enough to construct an effective remote enqueue operation with a single network transaction.

7.7 Shared Physical Address Space

In this section we examine a fourth major design style for the communication architecture of scalable multiprocessors – a shared physical address space. This builds directly upon the modest scale shared memory machines and provides the same communication primitives: loads, stores, and atomic operations on shared memory locations. Many machines have been developed to extend this approach to large scale systems, including CM*, C.mmp, NYU Ultracomputer, BBN Butterfly, IBM RP3, Denelcor HEP-1, BBN TC2000, and the Cray T3D. Most of the early designs employed a dancehall organization, with the interconnect between the memory and the processors, whereas most of the later designs distributed memory organization. The communica-tion assist translates bus transactions into network transactions. The network transactions are very specific, since they describe only a predefined set of memory operations, and are interpreted directly by communication assist at the remote node.
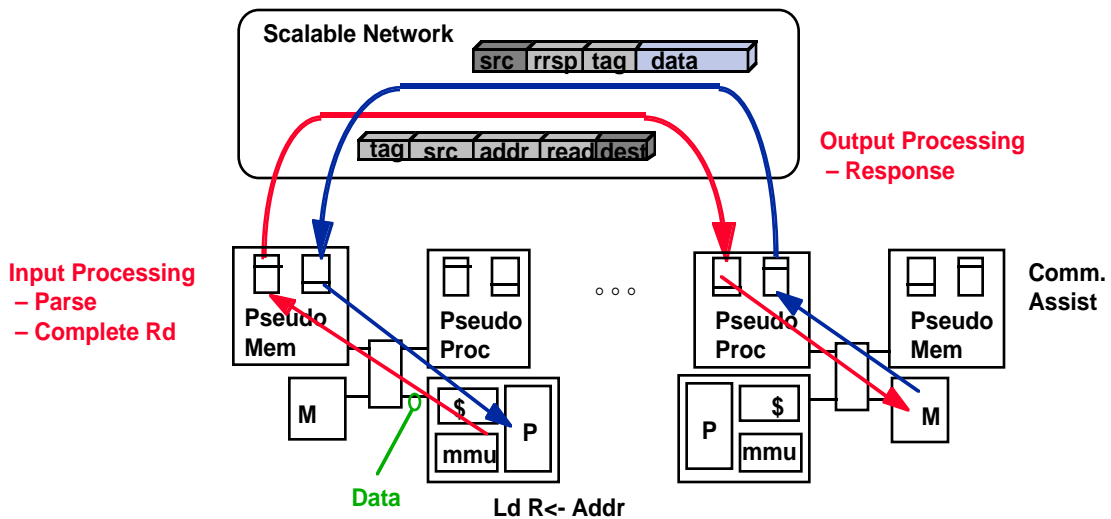
Figure 7-25 Shared physical address space machine organization

In scalable shared physical address machines, network transactions are initiated as a result of conventional memory instructions. They have a fixed set of formats, interpreted directly in the communication assist hardware. The oper-ations are request/response, and most systems provide two distinct networks to avoid fetch deadlock. The commu-nication architecture must assume the role of pseudo memory unit on the issuing node and pseudo processor on the servicing node. The remote memory operation is accepted by the pseudo memory unit on the issuing node, which carries out request-response transactions with the remote node. The source bus transaction is held open for the dura-tion of the request network transaction, remote memory access, and response network transaction. Thus, the com-munication assist must be able to access the local memory on the remote node even while the processor on that node is stalled in the middle of its own memory operation.

A generic machine organization for a large scale distributed shared physical address machine is shown in Figure 7-25. The communication assist is best viewed as forming a *pseudo-memory*

module and a *pseudo-processor*, integrated into the processor-memory connection. Consider, for example, a load instruction executed by the processor on one node. The on-chip memory management unit translates the virtual address into a global physical address, which is presented to the memory system. If this physical address is local to the issuing node, the memory simply responds with the contents of the desired location. If not, the communication assist must act like a memory module, while it access the remote location. The pseudo-memory controller accepts the read transaction on the memory bus, extracts the node number from the global physical address and issues a network transaction to the remote node to access the desired location. Note that at this point the load instruction is stalled between the address phase and data phase of the memory operation. The remote communication assist receives the network transaction, reads the desired location and issues a response transaction to the original node. The remote communication assist appears as a pseudo-processor to the memory system on its node when it issues the proxy read to the memory system. An important point to note is that when the pseudo-processor attempts to access memory on behalf of a remote node, the main processor there may be stalled in the middle of its own remote load instruction. A simple memory bus with a single outstanding operation is inadequate for the task. Either there must be two independent paths into memory or the bus must support split-phase operation with unordered completion. Eventually the response transaction will arrive at the originating pseudo-memory controller. It will complete the memory read operation just as if it were a (slow) memory module.

A key issue, which we will examine deeply in the next chapter, is the cachability of the shared memory locations. In most modern microprocessors, the cachability of an address is determined by a field in the page table entry for the containing page, which are extracted from the TLB when the location is accessed. In this discussion it is important to distinguish two orthogonal concepts. An address may be either private to a process or shared among processes, and it may be either physically local to a processor or physically remote. Clearly, addresses that are private to a process and physically local to the processor on which that process executes should be cached. This requires no special hardware support. Private data that is physically remote can also be cached, although this requires that the communication assists support cache block transactions, rather than just single words. There is no processor change required to cache remote blocks, since remote memory accesses appear identical to local memory accesses, only slower. If physically local and logically shared data is cached locally, then accesses on behalf of remote nodes performed by the pseudo-processor must be cache coherent. If local, shared data is cached only in write-through mode, this only requires that the pseudo-processor can invalidate cached data when it performs writes to memory on behalf of remote nodes. To cache shared data as write-back, the pseudo processor needs to be able to cause data to be flushed out of the cache. The most natural solution is to integrate the pseudo-processor on a cache coherent memory bus, but the bus must also be split-phase with some number of outstanding transactions reserved for the pseudo-processor. The final option is to cache shared, remote data. The hardware support for accessing, transferring, and placing a remote block in the local cache is completely covered by the above options. The new issue is keeping the possibly many copies of the block in various caches coherent. We must also deal with the consistency model for such a distributed shared memory with replication. These issues require substantially design consideration, and we devote all of Chapter 7 to addressing them. It is clearly attractive from a performance viewpoint to cache shared remote data that is mostly accessed locally

7.7.1 Case study: Cray T3D

The Cray T3D(R) provides a concrete example of an aggressive shared global physical address design using a modern microprocessor. The design follows the basic outline of Figure 7-13, with pseudo-memory controller and pseudo-processor providing remote memory access via a scalable network supporting independent delivery of request and response transactions. There are seven specific network transaction formats, which are interpreted directly in hardware. However, the design extends the basic shared physical address approach in several significant ways. The T3D system is intended to scale to 2048 nodes, each with a 150 MHz dual-issue DEC Alpha 21064 microprocessor and up to 64 MB of memory, as illustrated in Figure 7-26. The DEC Alpha architecture is intended to be used as a building block for parallel architectures [Alp92] and several aspects of the 21064 strongly influenced the T3D design. We first describe salient aspects of the microprocessor itself and the local memory system. Then we discuss the assists constructed around the basic processor to provide a shared physical address space, latency tolerance, block transfer, synchronization, and fast message passing. The Cray designers sometimes refer to this as the "shell" of support circuitry around a conventional microprocessor that embodies the parallel processing capability.
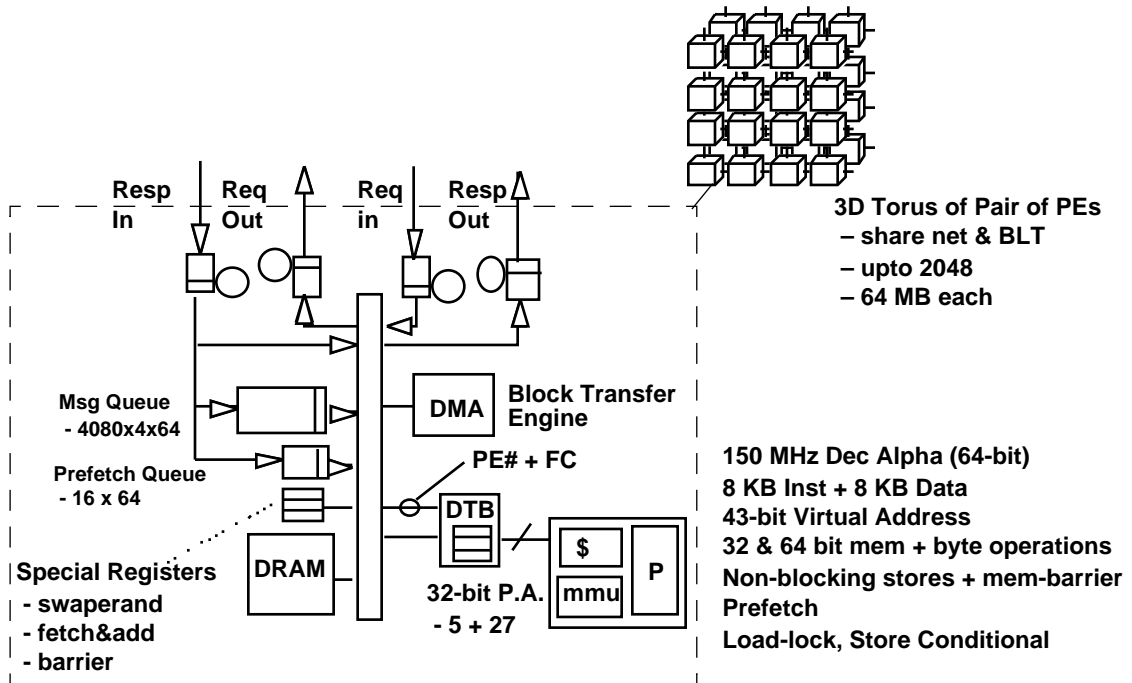
Figure 7-26 Cray T3D machine organization

Each node contains an elaborate communication assist, which includes more more than the pasedo memory and pseudo processor functions required for a shared physical address space. A set of external segment registers (the DTB) are provided to extend the machines limited physical address space. A prefetch queue support latency hiding through explicit read-ahead. A message queue supports events associated with message passing models. A collection of DMA units provide block transfer capability and special pointwise and global synchronization operations are supported. The machine is organized as a 3D torus, as the name might suggest.

The Alpha 21064 has 8KB on-chip instruction and data caches, as well as support for an external L2 cache. In the T3D design, the L2 cache is eliminated in order to reduce the time to access main memory. The processor stalls for the duration of a cache miss, so reducing the miss latency

directly increases the delivered bandwidth. The Cray design is biased toward access patterns typical of vector codes, which scan through large regions of memory. The measured access time of a load that misses to memory is 155 ns (23 cycles) on the T3D compared to 300 ns (45 cycles) on a DEC Alpha workstation at the same clock rate with a 512 KB L2 cache[Arp*95]. (The Cray T3D access time increases to 255 ns if the access is off-page within the DRAM.) These measurements are very useful in calibrating the performance of the global memory accesses, discussed below.

The Alpha 21064 provides a 43-bit virtual address space in accordance with the Alpha architecture, however, the physical address space is only 32 bits in size. Since the virtual-to-physical translation occurs on-chip, only the physical address is presented to the memory system and communication assist. A fully populated system of 2048 nodes with 64 MB each would require 37 bits of global physical address space. To enlarge the physical address space of each node, the T3D provides an external register set, called the DTB Annex, which uses 5 bits of the physical address to select a register containing a 21-bit node number; this is concatenated with a 27-bit local physical address to form a full global physical address.[1] The Annex registers also contain an additional field specifying the type of access, e.g., cached or uncached. Annex register 0 always refers to the local node. The Alpha load-lock and store-conditional instructions are used in the T3D to read and write the Annex registers. Updating an Annex register takes 23 cycles, just like an off-chip memory access, and can be followed immediately by a load or store instruction that uses the Annex register.

A read or write of a location in the global address space is accomplished by a short sequence of instructions. First, processor number part of the global virtual address is extracted and stored into an Annex register. A temporary virtual address is constructed so that the upper bits specify this Annex register and the lower bits the address on that node. Then a load or store instruction is issued on the temporary virtual address. The load operation takes 610 ns (91 cycles), not including the Annex setup and address manipulation. (This number increases by 100 ns (15 cycles) if the remote DRAM access is off-page. Also, if a cache block is brought over it increases to 785-885 ns.) Remember the virtual-to-physical translation occurs in the processor issuing the load. The page tables are set up so that the Annex register number is simply carried through from the virtual address to the physical address. So that the resulting physical address will make sense on the remote node, the physical placement of all processes in a parallel program is identical. (Paging is not supported). In addition, care is exercised to ensure that all processes in a parallel program have extended their heap to the same length.

The Alpha 21064 provides only non-blocking stores. Execution is allowed to proceed after a store instruction without waiting for the store to complete. Writes are buffered by write buffer, which is four deep and in each entry up to 32 bytes of write data can be merged. Several store instructions can be outstanding. A 'memory barrier' instruction is provided to ensure that writes have completed before further execution commences. The Alpha non-blocking store allows remote stores to be overlapped, so high bandwidth can be achieved in writes to remote memory locations. Remote writes of up to a full cache block can issue from the write buffer every 250 ns, providing up to 120 MB/s of transfer bandwidth from the local cache to remote memory. A single

---

1. This situation is not altogether unusual. The C.mmp employed a similar trick to overcome the limited addressing capability of the LSI-11 building block. The problems that arose from this led to a famous quote attributed variously to Gordon Bell or Bill Wulf that the only flaw in an architecture that is hard to overcome is too small an address space.

blocking remote-write involves a sequence to issue the store, push the store out of the write buffer with a memory barrier operation, and then a wait on a completion flag provided by the network interface. This requires 900 ns, plus the Annex setup and address arithmetic.

The Alpha provides a special "prefetch" instruction, intended to encourage the memory system to move important data closer to the processor. This is used in the T3D to hide the remote read latency. An off-chip prefetch queue of 16 words is provided. A prefetch causes a word to be read from memory and deposited into the queue. Reading from the queue pops the word at its head. The prefetch issue instruction is treated like a store and takes only a few cycles. The pop operation takes the 23 cycles typical of an off chip access. If eight words are prefetched and then popped, the network latency is completely hidden and the effective latency of each is less than 300 ns.

The T3D also provides a bulk transfer engine, which can move blocks or regular strided data between the local node and a remote node in either direction. Reading from a remote node, the block transfer bandwidth peaks at 140 MB/s and writing to a remote node it peaks at 90 MB/s. However, use of the block transfer engine requires a kernel trap to provide the virtual to physical translation. Thus, the prefetch queue provides better performance for transfers up to 64 KB of data and non-blocking stores are faster for any length. The primary advantage of the bulk transfer engine is the ability to overlap communication and computation. This capability is limited to some extent, since the processor and the bulk transfer engine compete for the same memory bandwidth.

The T3D communication assist also provides special support for synchronization. First, there is a dedicated network to support global-OR and global-AND operations, especially for barriers. This allows processors to raise a flag indicating that they have reached the barrier, continue executing, and then wait for all to enter before leaving. Each node also has a set of external synchronization registers to support atomic-swap and fetch&inc. There is also a user-level message queue, which will cause a message either to be enqueued or a thread invoked on a remote node. Unfortunately, either of these involve a remote kernel trap, so the two operations take 25 μs and 70 μs, respectively. In comparison, building a queue in memory using the fetch&inc operation allows a four word message to be enqueued in 3μs and dequeued in 1.5μs.

7.7.2 Cray T3E

The Cray T3E [Sco96] follow-on to the Cray T3D provides an illuminating snapshot of the trade-offs in large scale system design. The two driving forces in the design were the need to provide a more powerful, more contemporary processor in the node and to simplify the "shell." The Cray T3D has many complicated mechanisms for supporting similar functions, each with unique advantages and disadvantages. The T3E uses the 300 MHz, quad-issue Alpha 21164 processor with a sizable (96 KB) second-level on-chip cache. Since the L2 cache is on chip, eliminating it is not an option as on the T3D. However, the T3E forgoes the board-level tertiary cache typically found in Alpha 21164 based workstations. The various remote access mechanisms are unified into a single external register concept. In addition, remote memory access are performed using virtual addresses that are translated to physical addresses by the remote communication assist.

A user process has access to a set of 512 E-registers of 64 bits each. The processor can read and write contents of the E-registers using conventional load and store instructions to a special region of the memory space. Operations are also provided to get data from global memory into an E-reg-

ister, to put data from an E-register to global memory, and to perform atomic read-modify-writes between E-registers and global memory. To load remote data into an E-register involves three steps. First, the processor portion of the global virtual address is constructed in an E-address register. Second, the get command issued via a store to a special region of memory. The field of the address used in the command store specifies the put operation and another field specifies the destination data E-register. The command-store data specifies an offset to be used relative to the address E-register. The command store has the side-effect of causing the remote read to be performed and the destination E-register to be loaded with the data. Finally, the data is read into the processor via a load to the data E-register. The process for a remote put is similar, except that the store data is placed in the data E-register, which is specified in the put command-store. This approach of causing load and stores to data registers as a side-effect of operations on address registers goes all the way back to the CDC-6600[Tho64], although it seems to have been largely forgotten in the meanwhile.

The utility of the prefetch queue is provided by E-registers by associating a full/empty bit with each E-register. A series of gets can be issued, and each one sets the associated destination E-register to empty. When the gets complete, the register is set to full. If the processor attempts to load from an empty E-register, the memory operation is stalled until the get completes. The utility of the block transfer engine is provided by allowing vectors of four or eight words to be transferred through E-registers in a single operation. This has the added advantages of providing a means of efficient gather operations.

The improvements in the T3E greatly simplify code generation for the machine and offer several performance advantages, however, these are not at all uniform. The computational performance is significantly higher on the T3D due to the faster processor and larger on-chip cache. However, the remote read latency more than twice that of the T3D, increasing from 600 ns to roughly 1500 ns. The increase is due to the level 2 cache miss penalty and the remote address translation. The remote write latency is essentially the same. The prefetch cost is improved by roughly a factor of two, obtaining a rate of one word read every 130 ns. Each of the memory modules can service a read every 67 ns. Non-blocking writes have essentially the same performance on the two machines. The block transfer capability of the T3E is far superior to the T3D. A bandwidth of greater than 300 MB/s is obtained without the large startup cost of the block transfer engine. The bulk write bandwidth is greater than 300 MB/s, three times the T3D.

7.7.3 Summary

There is a wide variation in the degree of hardware interpretation of network transactions in modern large scale parallel machines. These variations result in a wide range in the overhead experienced by the compute processor in performing communication operations, as well as in the latency added to that of the actual network by the communication assist. By restricting the set of transactions, specializing the communication assist to the task of interpreting these transactions, and tightly integrating the communication assist with the memory system of the node, the overhead and latency can be reduced substantially. The hardware specialization also can provide the concurrency need within the assist to handle several simultaneous streams of events with high bandwidth.

7.8 Clusters and Networks of Workstations

Along with the use of commodity microprocessors, memory devices, and even workstation operating systems in modern large-scale parallel machines, scalable communication networks similar to those used in parallel machines have become available for use in a limited local area network setting. This naturally raises the question as to what extent are networks of workstations (NOWs) and parallel machines are converging. Before entertaining this question a little background is in order.

Traditionally, collections of complete computers with a dedicated interconnect, often called *clusters*, have been used to serve multiprogramming workloads and to provide improved availability[Kro*86,Pfi95]. In multiprogramming clusters, a single front-end machine usually acts as an intermediary between a collection of compute servers and a large number of users at terminals or remote machines. The front-end tracks the load on the cluster nodes and schedules tasks onto the most lightly loaded nodes. Typically, all the machines in the cluster are setup to function identically; they have the same instruction set, the same operating systems, and the same file system access. In older systems, such as the Vax VMS cluster[Kro*86], this was achieved by connecting each of the machines to a common set of disks. More recently, this *single system image* is usually achieved by mounting common file systems over the network. By sharing a pool of functionally equivalent machines, better utilization can be achieved on a large number of independent jobs.

Availability clusters seek to minimize downtime of large critical systems, such as important on-line databases and transaction processing systems. Structurally they have much in common with multiprogramming clusters. A very common scenario is to use a pair of SMPs running identical copies of a database system with a shared set of disks. Should the primary system fail due to a hardware or software problem, operation rapidly "fails over" to the secondary system. The actual interconnect that provides the shared disk capability can be dual access to the disks or some kind of dedicated network.

A third major influence on clusters has been the rise of popular public domain software, such as Condor[Lit*88] and PVM[Gei93], that allow users to farm jobs over a collection of machines or to run a parallel program on a number of machines connected by an arbitrary local-area or even wide-area network. Although the communication performance capability is quite small, typical latencies are a millisecond or more for even small transfers and the aggregate bandwidth is often less than one MB/s, these tools provide an inexpensive vehicle for a class of problems with a very high ratio of computation to communication.

The technology breakthrough that presents the potential of clusters taking on an important role in large scale parallel computing is a scalable, low-latency interconnect, similar in quality to that available in parallel machines, but deployed like a local-area network. Several potential candidates networks have evolved from three basic directions. Local area networks have traditionally been either a shared bus (e.g. Ethernet) or a ring (e.g., Token Ring and FDDI) with fixed aggregate bandwidth, or a dedicated point-to-point connection (e.g., HPPI). In order to provide scalable bandwidth to support a large number of fast machines, there has been a strong push toward switch based local area network (e.g., HPPI switches, FDDI switches[LuPo97], and Fiberchannels. Probably the most significant development is the widespread adoption of the ATM (asynchronous transfer mode) standard, developed by the Telecommunications industry, as a switched LAN. Several companies offer ATM switches (routers in the terminology of Section 7.2) with up to sixteen ports at 155 Mb/s (19.4 MB/s) link bandwidth. These can be cascaded to form a larger

network. Under ATM, a variable length message is transferred on a pre-assigned route, called "virtual circuit", as a sequence of 53 byte cells (48 bytes of data and 5 bytes of routing information). We will look at these networking technologies in more detail in Chapter 8. In terms of the model developed in Section 7.2, current ATM switches typically have a routing delay of about 10 μs in an unloaded network, although some are much higher. A second major standardization effort is represented by SCI (scalable coherent interconnect), which includes a physical layer standard and a particular distributed cache coherency strategy.

There has also been a strong trend to evolve the proprietary networks used within MPP systems into a form that can be used to connect a large number of independent workstations or PCs over a sizable area. Examples of this include Tnet[Hor95] (now System Net) from Tandem Corp. and Myrinet[Bod*95]. The Myrinet switch provides eight ports at 160 MB/s each which can be cascaded in regular and irregular topologies to form a large network. It transfers variable length packets with a routing delay of about 350 ns per hop. Link level flow control is used to avoid dropping packets in the presence of contention.

As with more tightly integrated parallel machines, the hardware primitives in emerging NOWs and Clusters remains an open issue and subject to much debate. Conventional TCP/IP communication abstractions over these advanced networks exhibit large overheads (a millisecond or more)[Kee*95], in many cases larger than that of common ethernet. A very fast processor is required to move even 20 MB/s using TCP/IP. However, the bandwidth does scale with the number of processors, at least if there is little contention. Several more efficient communication abstractions have been proposed, including Active Messages[And*95,vEi*92,vEi*95] and Reflective Memory[Gil96, GiKa97]. Active Messages provide user-level network transactions, as discussed above. Reflective Memory allows writes to special regions of memory to appear as writes into regions on remote processors; there is no ability to read remote data, however. Supporting a true shared physical address space in the presence of potential unreliability, i.e., node failures and network failures, remains an open question. An intermediate strategy is to view the logical network connecting a collection of communicating processes as a fully connected group of queues. Each processes has a communication endpoint consisting of a send queue, a receive queue, and a certain amount of state information, such as whether notifications should be delivered on message arrival. Each process can deliver a message to any of the receive queues by depositing it in its send queue with an appropriate destination identifier. At the time of writing, this approach is being standardized by an industry consortium, lead by Intel, Microsoft, and Compaq, as the Virtual Interface Architecture, based on several research efforts, including Berkeley NOW, Cornell UNET, Illinois FM, and Priceton SHRIMP projects.

The hardware support for the communication assists and the interpretation of the network transactions within clusters and NOWs spans almost the entire range of design points discussed in Section 7.3.5. However, since the network plugs into existing machines, rather than being integrated into the system at the board or chip level, typically it must interface at an I/O bus rather than at the memory bus or closer to the processor. In this area too there is considerable innovation. Several relatively fast I/O busses have been developed which maintain cache coherency. The most notable development being PCI. Experimental efforts have integrated the network through the graphics bus[Mar*93].

An important technological force further driving the advancement of clusters is the availability of relatively inexpensive SMP building blocks. For example, clustering a few tens of Pentium Pro "quad pack" commodity servers yields a fairly large scale parallel machine with very little effort. At the high-end, most of the very large machine are being constructed as a highly optimized clus-

ter of the vendor's largest commercially available SMP node. For example, in the 1997-8 window of machines purchased by the Department of Energy as part of the Accelerated Strategic Computing Initiative, the Intel machine is built as 4536 dual-processor Pentium Pros. The IBM machine is to be 512 4-way PowerPC 604s, upgraded to 8-way PowerPC 630s. The SGI/Cray machine is initially sixteen 32-way Orgins interconnected with many HPPI 6400 links, eventually to be integrated into larger cache coherent units, as described in the next chapter.

7.8.1 Case Study: Myrinet SBus Lanai

A representative example of an emerging NOW is illustrated by Figure 7-27. A collection of ULTRASparc workstations are integrated using a Myrinet scalable network via an intelligent network interface card (NIC). Let us start with the basic hardware operation and work upwards. The network illustrates what is becoming to be known as a *system area network* (SAN), as opposed to a tightly packaged parallel machine network or widely dispersed local area network (LAN). The links are parallel copper twisted pairs (18 bits wide) and can be a few tens of feet long, depending on link speed and cable type. The communication assist follows the dedicated message processor approach similar to the Meiko CS2 and IBM SP2. The NIC contains a small embedded "Lanai" processor to control message flow between the host and the network. A key difference in the Myricom design is that the NIC contains a sizable amount of SRAM storage. All message data is staged through NIC memory between the host and the network. This memory is also used for Lanai instruction and data storage. There are three DMA engines on the NIC, one for network input, one network output, and one for transfers between the host and the NIC memory. The host processor can read and write NIC memory using conventional loads and stores to properly regions of the address space, i.e., through programmed I/O. The NIC processor uses DMA operations to access host memory. The kernel establishes regions of host memory that are accessible to the NIC. For short transfers it is most efficient for the host to move the data directly into and out of the NIC, whereas for long transfers it is better for the host to write addresses into the NIC memory and for the NIC to pick up these addresses and use them to set up DMA transfers. The Lanai processor can read and write the network FIFOs or it can drive them by DMA operations from or to NIC memory.

The "firmware" program executing within the Lanai primarily manages the flow of data by orchestrating DMA transfers in response to commands written to it by the host and packet arrivals from the network. Typically, a command is written into NIC memory, where is it picked up by the NIC processor. The NIC transfers data, as required, from the host and pushes it into the network. The Myricom network uses source based routing, so the header of the packet includes a simple routing directive for each network switch along the path to the destination. The destination NIC receives the packet into NIC memory. It can then inspect the information in the transaction and process it as desired to support the communication abstraction.

The NIC is implemented as four basic components, a bus interface FPGA, a link interface, SRAM, and the Lanai chip, which contains the processor, DMA engines and link FIFOs. The link interface converts from on-board CMOS signals to long-line differential signalling over twisted pairs. A critical aspect of the design is the bandwidth to the NIC memory. The three DMA engines and the processor share an internal bus, implemented within the Lanai chip. The network DMA engines can demand 320 MB/s, while the host DMA can demand short bursts of 100 MB/s on an SBus or long bursts of 133 MB/s on a PCI bus. The design goal for the firmware to keep all three DMA engines active simultaneously, however, this is a little tricky because once it starts the
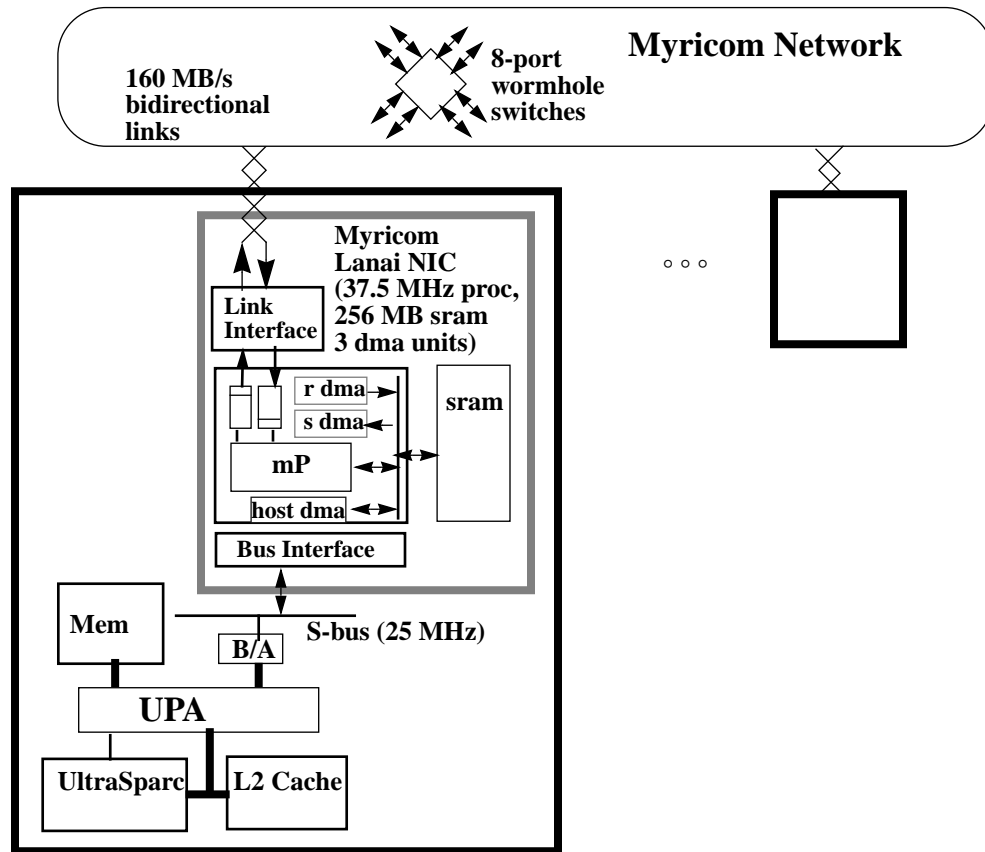
Figure 7-27 NOW organization using Myrinet and dedicated message processing with an embedded processor

Although the nodes of a cluster are complete conventional computers, a sophisticated communication assist can be provided within the interface to a scalable, low latency network. Typically, the network interface is attached to a conventional I/O bus, but increasingly vendors are providing means of tighter integration with the node architecture. In many cases, the communication assist provides dedicated processing of network transactions.

DMA engines its available bandwidth and hence its execution rate are reduce considerably by competition of the SRAM.

Typically, the NIC memory is logically divided into a collection of functionally distinct regions, including instruction storage, internal data structures, message queues, and data transfer buffers. Each page of the NIC memory space is independently mapped by the host virtual memory system. Thus, only the kernel can access the NIC processor code and data space. The remaining communication space can be partitioned into several, disjoint communication regions. By controlling the mapping of these communication regions, several user processes can have *communication endpoints* that are resident on the NIC, each containing message queues and associated data transfer area. In addition, a collection of host memory frames can be mapped into the IO space accessible from the NIC. Thus, several user processes can have the ability to write messages into the NIC and read messages directly from the NIC, or to write message descriptors containing pointers to data that is to be DMA transferred through the card. These communication end points can be managed much like conventional virtual memory, so writing into an endpoint

causes it to be made resident on the NIC. The NIC firmware is responsible for multiplexing messages from the collection of resident endpoints onto the actual network wire. It detects when a message has been written into a send queue and forms a packet by translating the users destination address into a route through the network to the destination node and an indentifier of the endpoint on that node. In addition, it places a source identifier in the header, which can be checked at the destination. The NIC firmware inspects the header for each incoming packet. If it is destined for a resident endpoint, then it can be deposited directly in the associated receive buffer and, optionally, for a bulk data transfer if the destination region is mapped the data can be DMA transferred into host memory. If these conditions are not met, or if the message is corrupted or the protection check violated, the packet can be NACKed to the source. The driver that manages the mapping of endpoints and data buffer spaces is notified to cause the situation to be remedied before the message is successfully retried.

7.8.2   Case Study: PCI Memory Channel

A second important representative cluster communication assist design is the Memory Channel[Gil96] developed by Digital Equipment Corporation, based on the Encore Reflective memory and research efforts in the virtual memory mapped communication[Blu*94,Dub*96]. This approach seeks to provide a limited form of a shared physical address space without needing to fully integrate the pseudo memory device and pseudo processor into the memory system of the node. It also preserves some of the autonomy and independent failure characteristics of clusters. As with other clusters, the communication assist is contained in a network interface card that is inserted into a conventional node on an extension bus, in this case the PCI (peripheral connection interface) I/O bus.

The basic idea behind reflective memory is to establish a connection between a region of the address space of one process, a 'transmit region', and a 'receive region' in another, as indicated by Figure 7-28. Data written to a transmit region by the source is "reflected" into the receive region of the destination. Usually, a collection of processes will have a fully connected set of transmit-receive region pairs. The transmit regions on a node are allocated from a portion of the physical address space that is mapped to the NIC. The receive regions are locked-down in memory, via special kernel calls, and the NIC is configured so that it can DMA transfer data into them. In addition, the source and destination processes must establish a connection between the source transmit region and the destination receive region. Typically, this is done by associating a key with the connection and binding each region to the key. The regions are an integral number of pages.

The DEC memory channel is a PCI-based NIC, typically placed in a Alpha based SMP, described by Figure 7-29[Gil96]. In addition to the usual transmit and receive FIFOs, it contains a page control table (PCT), a receive DMA engine, and transmit and receive controllers. A block of data is written to the transmit region with a sequence of stores. The Alpha write buffer will attempt to merge updates to a cache block, so the send controller will typically see a cache-block write operation. The upper portion of the address given to the controller is the frame number, which is used to index into the PCT to obtain a descriptor for the associated receive region, i.e., the destination node (or route to the node), the receive frame number, and associated control bits. This information, along with source information, is placed into the header of the packet that is delivered to the destination node through the network. The receive controller extracts the receive frame number and uses it to index into the PCT. After checking the packet integrity and verifying the source, the data is DMA transferred into memory. The receive regions can be cachable host
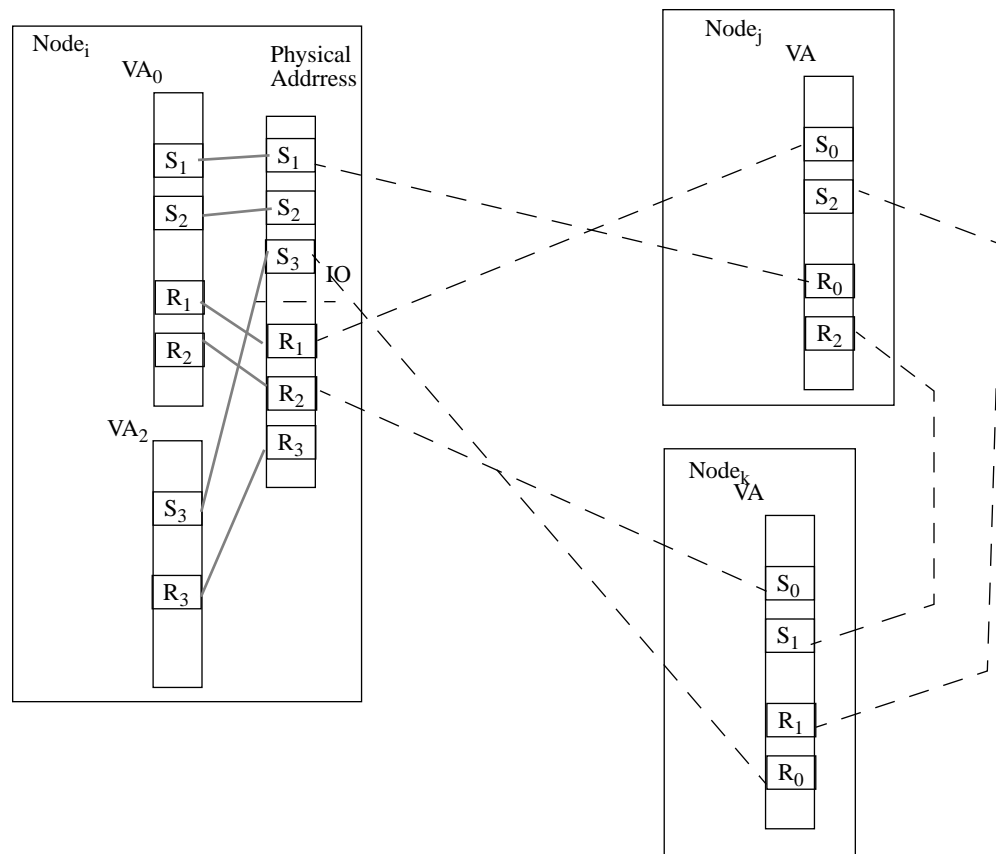
Figure 7-28 Typical Reflective Memory address space organization

Send regions of the processess on a node are mapped to regions in the physical address space associated with the NIC. Receive regions are pinned in memory and mappings within the NIC are established so that it can DMA the associated data to host memory. Here, node$_i$ has two communicating processes, one of which has established reflective memory connections with processes on two other node. Writes to a send region generate memory transactions against the NIC. It accepts the write data, builds a packet with a header indentifying the receive page and offset, and routes the packet to the destination node. The NIC,upon accepting a packet from the network inspects the header and DMA transfers the data into the corresponding receive page. Optionally, packet arrival may generate an interrupt. In general, the user process scans receive regions for relavant updates. To support message passing, the receive pages contain message queues.

memory, since the transfers across the memory bus are cache coherent. If required, an interrupt is raised upon write completion.

As with a shared physical address space, this approach allows data to be transferred between nodes by simply storing the data from the source and loading it at the destination. However, the use of the address space is much more restrictive, since data that is to be transferred to a particular node must be placed in a specific send page and receive page. This is quite different from the scenario where any process can write to any shared address and any process can read that address. Typically, shared data structures are not placed in the communication regions. Instead the regions are used as dedicated message buffers. Data is read from a logically shared data structure and transmitted to a process that requires it through a, logical, memory channel. Thus, the communication abstraction is really one of memory-based message passing. There is no mechanism to read a remote location; a process can only read the data that has been written to it. To bet-
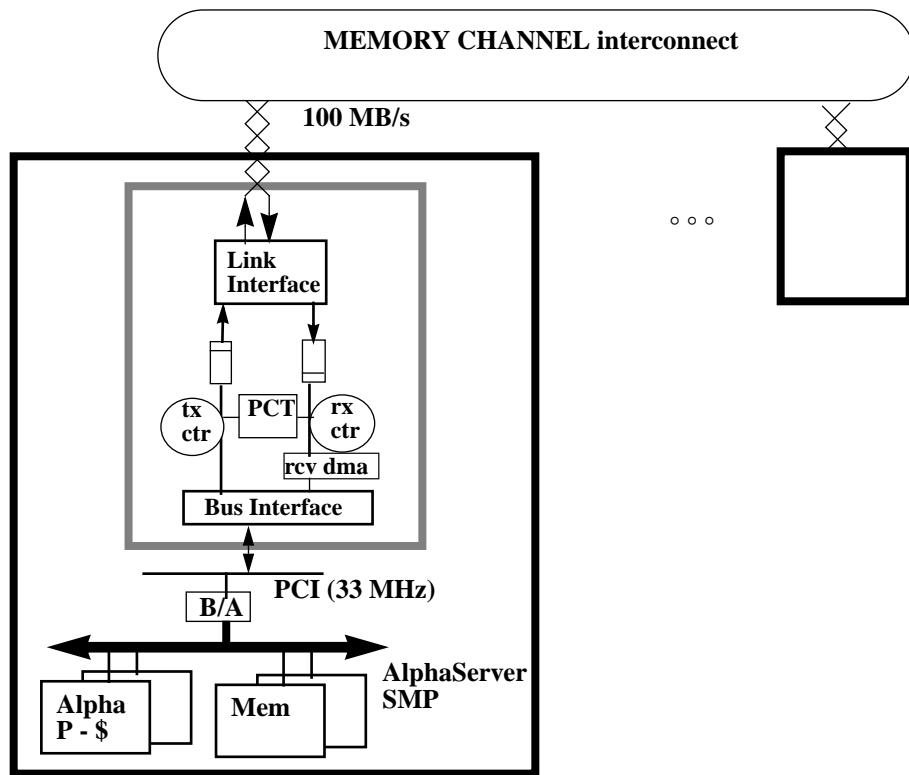
Figure 7-29 DEC Memory Channel Hardware Organization

A MEMORY CHANNEL cluster consists of a collection of AlphaServer SMPs with PCI MEMORY CHANNEL adapters. The adapter contains a page control table which maps local transmit regions to remote receive regions and local receive regions to locked down physical page frames. The transmit controller (tx ctr) accepts PCI write transactions, constructs a packet header using the store address and PCT entry contents, and deposits a packet containing the write data into the transmit FIFO. The receive controller (rx ctrl) checks the CRC and parses the header of an incoming packet before initiating a DMA transfer of the packet data into host memory. Optionally, an interrupt may be generated. In the initial offering, the memory channel interconnect is a shared 100 MB/s bus, but this is intended to be replaced by a switch. Each of the nodes is typically a sizable SMP AlphaServer.

ter support the "write by one, read by any" aspect of shared memory, the DEC memory channel allows transmit regions to multicast to a group of receive regions, including a loop-back region on the source node. To ease construction of distributed data structures, in particular a distributed lock manager, the FIFO order is preserved across operations from a transmit region. Since reflective memory builds upon, rather than extends the node memory system, the write to a transmit region finishes as soon as the NIC accepts the data. To determine that the write has actually occurred, the host checks a status register in the NIC. The raw MEMORY CHANNEL interface obtains an one-way communication latency of 2.9 μs and a transfer bandwidth of 64 MB/s between two 300 MHz DEC AlphaServers.[Law*96] The one way latency for a small MPI message is about 7 μs and a maximum bandwidth of 61 MB/s is achieved on long messages. Using the TruCluster MEMORY CHANNEL software[Car*96], acquiring and releasing and uncontended spinlock takes approximately 130 μs and 120 μs, respectively.

The Princeton SHRIMP designs[Blu*94,Dub*96] extend the reflective memory model to better support message passing. They allow the receive offset to be determined by a register at the des-

tination, so successive packets will be queued into the receive region. In addition, a collection of writes can be performed to a transmit region and then a segment of the region transmitted to the receive region.

7.9 Comparison of Communication Performance

Now that we have seen the design spectrum of modern scalable distributed memory machines, we can solidify our understanding of the impact of these design trade-offs in terms of the communication performance that is delivered to applications. In this section we examine communication performance through microbenchmarks at three levels. The first set of microbenchmarks use a non-standard communication abstraction that closely approximates the basic network transaction on a user-to-user basis. The second use the standard MPI message passing abstraction and the third a shared address space. In making this comparison, we can see the effects of the different organizations and of the protocols used to realize the communication abstractions.

7.9.1 Network Transaction Performance

Many factors interact to determine the end-to-end latency of the individual network transaction, as well as the abstraction built on top of it. When we measure the time per communication operation we observe the cumulative effect of these interactions. In general, the measured time will be larger than what one would obtain by adding up the time through each of the individual hardware components. As architects, we may want to know how each of the components impacts performance, however, what matters to programs is the cumulative effect, including the subtle interactions which inevitably slow things down. Thus, in this section we take an empirical approach to determining the communication performance of several of our case study machines. We use as a basis for the study a simple user-level communication abstraction, called Active Messages, which closely approximates the basic network transaction. We want to measure not only the total message time, but the portion of this time that is overhead in our data transfer equation (Equation 1.5) and the portion that is due to occupancy and delay.

Active Messages constitute request and response transactions in a form which is essentially a restricted remote procedure call. A request consists of the destination processor address, an indentifier for the message handler on that processor, and a small number of data words in the source processor registers which are passed as arguments to the handler. An optimized instruction sequence issues the message into the network via the communication assist. On the destination processor an optimized instruction sequence extracts the message from the network and invokes the handler on the message data to perform a simple action and issue a response, which identifies a response handler on the original source processor. To avoid fetch deadlock without arbitrary buffering, a node services response handlers whenever attempting to issue an Active Message. If the network is backed up and the message cannot be issued, the node will continue to drain incoming responses. While issuing a request, it must also service incoming requests. Notification of incoming messages may be provided through interrupts, through signalling a thread, through explicitly servicing the network with a null message event, or through polling.

The microbenchmark that uses Active Messages is a simple echo test, where the remote processor is continually servicing the network and issuing replies. This eliminates the timing variations that would be observed if the processor was busy doing other work when the request arrived.

Also, since our focus is on the node-to-network interface, we pick processors that are adjacent in the physical network. All measurements are performed by the source processor, since many of these machines do not have a global synchronous clock and the "time skew" between processors can easily exceed the scale of an individual message time. To obtain the end-to-end message time, the round-trip time for a request-response is divided by two. However, this one-way message time has three distinct portions, as illustrated by Figure 7-30. When a processor injects a message, it is occupied for a number of cycles as it interfaces with the communication assist. We call this the *send overhead*, as it is time spent that cannot be used for useful computation. Similarly, the destination processor spends a number of cycles extracting or otherwise dealing with the message, called the *receive overhead*. The portion of the total message cost that is not covered by overhead is the *communication latency*. In terms of the communication cost expression developed in Chapter xx3, it includes the portions of the transit latency, bandwidth and assist occupancy components that are not overlapped with send or receive overhead. It can potentially be masked by other useful work or by processing additional messages, as we will discuss in detail in Chapter 8. From the processor's viewpoint, it cannot distinguish time spent in the communication assists from that spent in the actual links and switches of the interconnect. In fact, the communication assist may begin pushing the message into the network while the source processor is still checking status, but this work will be masked by the overhead. In our microbenchmark, we seek to determine the length of the three portions.
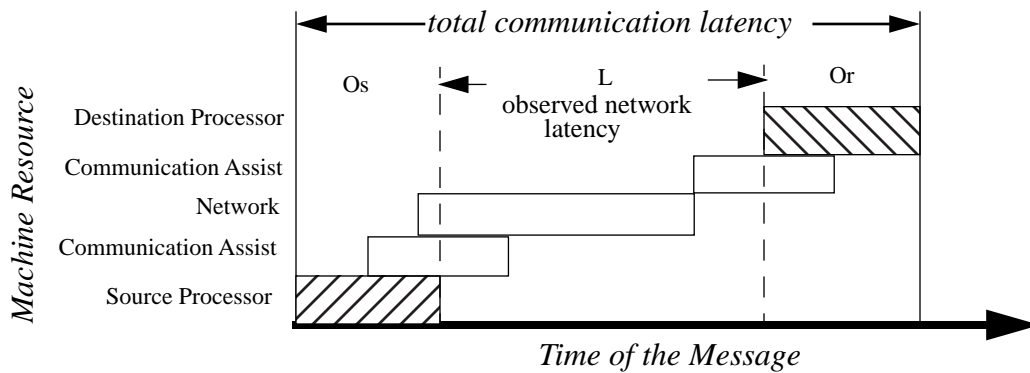
Figure 7-30 Breakdown of message time into source-overhead, network time, and destination overhead

A graphical depiction of the machine operation associated with our basic data transfer model for communication operations. The source processor spends O_s cycles injecting the message to the communication assist, during which time it can perform no other useful work, and similarly the destination experiences O_r overhead in extracting the message. To actually transfer the information involves the communication assists and network interfaces, as well as the links and switches of the network. As seen from the processor, these subcomponents are indistinquishable. It experiences the portion of the transfer time that is not covered by its own overhead as latency, that can be overlapped with other useful work. The processor also experiences the maximum message rate, which we represent in terms of the minimum average time between messages, or gap.

The left hand graph in Figure 7-31 shows a comparison of one-way Active Message time for the Thinking Machines CM5 (Section 7.5.1) Intel Paragon (Section 7.6.1) the Meiko CS-2 (Section 7.6.2) and a cluster of workstations (Section 7.8.1). The bars show the total one-way latency of a small (five word) message divided up into three segments indicating the processing overhead on the sending side (Os) the processing overhead on the receiving side (Or), and the

remaining communication latency (L). The bars on the right (g) shows the time per message for a pipelined sequence of request-response operations. For example, on the Paragon an individual message has an end-to-end latency of about 10 μs, but a burst of messages can go at about 7.5 μs per message, or a rate of $\dfrac{1}{7.5\mu s}$ = 133 thousand messages per second. Let us examine each of these components in more detail.

The send overhead is nearly uniform over the four designs, however, the factors determining this component are different on each system. On the CM-5, this time is dominated by uncached writes of the data plus the uncached read of the NI status to determine if the message was accepted. The status also indicates whether an incoming message has arrived, which is convenient, since the node must receive messages even while it is unable to send. Unfortunately, two status reads are required for the two networks. (A later model of the machine, the CM-500, provided more effective status information and substantially reduced the overhead.) On the Paragon, the send overhead is determined by the time to write the message into the memory buffer shared by the compute processor and message processor. The compute processor is able to write the message and set the 'message present' flag in the buffer entry with two quad word stores, unique to the i860, so it is surprising that the overhead is so high. The reason has to do with the inefficiency of the bus-based cache coherence protocol within the node for such a producer-consumer situation. The compute processor has to pull the cache blocks away from the message processor before refilling them. In the Meiko CS-2, the message is built in cached memory and then a pointer to the message (and command) is enqueued in the NI a single swap instruction. This instruction is provided in the Sparc instruction set to support synchronization operations. In this context, it provides a way of passing information to the NI and getting status back indicating whether the operation was successful. Unfortunately, the exchange operation is considerably slower than the basic uncached memory operations. In the NOW system, the send overhead is due to a sequence of uncached double word stores and an uncached load across the I/O bus to NI memory. Surprisingly, these have the same effective cost as the more tightly integrated designs.

An important point revealed by this comparison is that the cost of uncached operations, misses, and synchronization instructions, generally considered to be infrequent events and therefore a low priority for architectural optimization, are critical to communication performance. The time spent in the cache controllers before they allow the transaction to delivered to the next level of the storage hierarchy dominates even the bus protocol. The comparison of the receive overheads shows that cache-to-cache transfer from the network processor to the compute processor is more costly than uncached reads from the NI on the memory bus of the CM-5 and CS-2. However, the NOW system is subject to the greater cost of uncached reads over the I/O bus.

The latency component of the network transaction time is the portion of the one-way transfer that is not covered by either send or receive overhead. Several facets of the machine contribute to this component, including the processing time on the communication assist or in the network interface, the time to channel the message onto a link, and the delay through the network. Different facets are dominant in our study machines. The CM-5 links operate at 20 MB/s (4 bits wide at 40 MHz). Thus, the occupancy of a single wire to transfer a message with 40 bytes of data (the payload) plus an envelope containing route information, CRC, and message type is nearly 2.5 μs. With the fat-tree network, there are many links (typically N/4) crossing the top of the network, so the aggregate network bandwidth is seldom a bottleneck. Each router adds a delay of roughly 200 ns, and there are at most $2\log_4 N$ hops. The network interface occupancy is essentially the same as the wire occupancy, as it is a very simple device that spools the packet onto or off the wire.
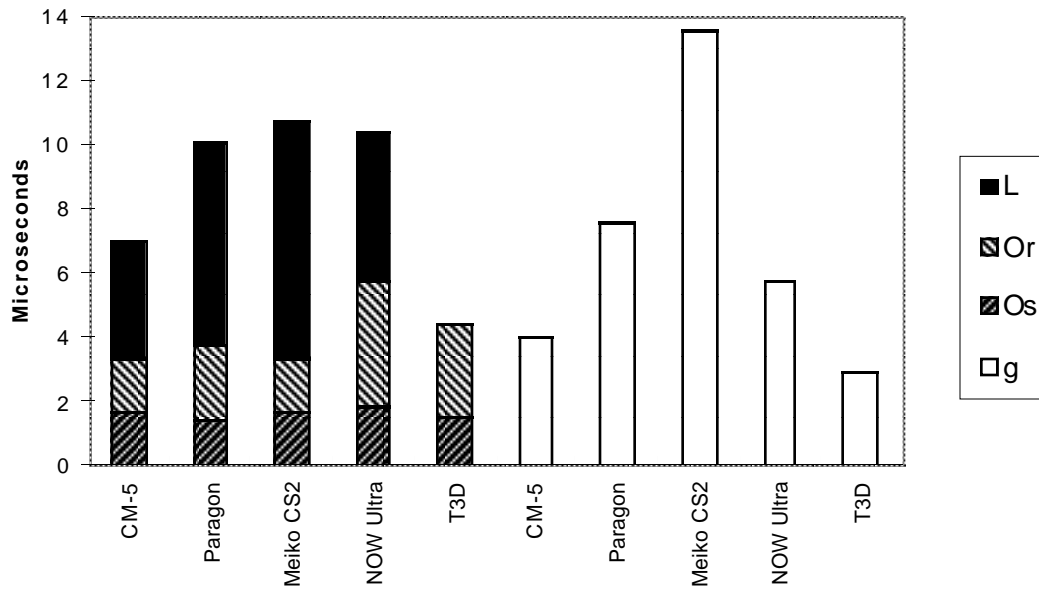
Figure 7-31 Performance comparison at the Network Transaction Level via Active Messages

In the Paragon, the latency is dominated by processing in the message processor at the source and destination. With 175 MB/s links, the occupancy of a link is only about a 300 ns. The routing delay per hop is also quite small, however, in a large machine the total delay may be substantially larger than the link occupancy, as the number of hops can be $2\sqrt{N}$. Also, the network bisection can potentially become a bottleneck, since only $\sqrt{N}$ links cross the middle of the machine. However, the dominant factor is the assist (message processor) occupancy in writing the message across the memory bus into the NI at the source and reading the message from the destination. These steps account for 4 μs of the latency. Eliminating the message processors in the communication path on the Paragon decreases the latency substantially. Surprisingly, it does not increase the overhead much; writing and reading the message to and from the NI have essentially the same cost as the cache to cache transfers. However, since the NI does not provide sufficient interpretation to enforce protection and to ensure that messages move forward through the network adequately, avoiding the message processors is not a viable option in practice.

The Meiko has a very large latency component. The network accounts for a small fraction of this, as it provides 40 MB/s links and a fat-tree topology with ample aggregate bandwidth. The message processor is closely coupled with the network, on a single chip. The latency is almost entirely due to accessing system memory from the message processor. Recall, the compute processor provides the message processor with a pointer to the message. The message processor performs DMA operations to pull the message out of memory. At the destination it writes it into a shared queue. An unusual property of this machine is that a circuit through the network is held open through the network transaction and an acknowledgment is provided to the source. This mechanism is used to convey to the source message processor whether it successfully obtained a

lock on the destination incoming message queue. Thus, even though the latency component is large, it is difficult to hide it through pipelining communication operations, since source and destination message processors are occupied for the entire duration.

The latency in the NOW system is distributed fairly evenly over the facets of the communication system. The link occupancy is small with 160 MB/s links and the routing delay is modest, with 350 ns per hop in roughly a fat tree topology. The data is deposited into the NI by the host and accessed directly from the NI. Time is spent on both ends of the transaction to manipulate message queues and to perform DMA operations between NI memory and the network. Thus, the two NIs and the network each contribute about a third of the 4 μs latency.

The Cray T3D provides hardware support for user level messaging in the form of a per processor message queue. The capability in the DEC Alpha to extend the instruction set with privileged subroutines, called PAL code, is used. A four-word message is composed in registers and a PAL call issued to send the message. It is placed in a user level message queue at the destination processor, the destination processor is interrupted, and control is returned either the user application thread, which can poll the queue, or to a specific message handler thread. The send overhead to inject the message is only 0.8 μs, however, the interrupt has an overhead of 25 μs and the switch to the message handler has a cost of 33 μs[Arp*95]. The message queue, without interrupts or thread switch, can be inserted into using the fetch&increment registers provided for atomic operations. The fetch&increment to advance the queue pointer and writing the message takes 1.5 μs, whereas dispatching on the message and reading the data, via uncached reads, takes 2.9 μs.

The Paragon, Meiko, and NOW machines employ a complete operating system on each node. The systems cooperate using messages to provide a single system image and to run parallel programs on collections of nodes. The message processor is instrumental in multiplexing communication from many user processes onto a shared network and demultiplexing incoming network transactions to the correct destination processes. It also provides flow control and error detection. The MPP systems rely on the physical integrity of a single box to provide highly reliable operation. When a user process fails, the other processes in its parallel program are aborted. When a node crashes, its partition of the machine is rebooted. The more loosely integrated NOW must contend with individual node failures that are a result of hardware errors, software errors, or physical disconnection. As in the MPPs, the operating systems cooperate to control the processes that form a parallel program. When a node fails, the system reconfigures to continue without it.

7.9.2 Shared Address Space Operations

It is also useful to compare the communication architectures of the case study machines for shared address space operations: read and write. These operations can easily be built on top of the user level message abstraction, but this does not exploit the opportunity to optimize for these simple, frequently occurring operations. Also, on machines where the assist does not provide enough interpretation the remote processor is involved whenever its memory is accessed. In machines with a dedicated message processor, it can potentially service memory requests on behalf of remote nodes, without involving the compute processor. On machines supporting a shared physical address, this memory request service is provided directly in hardware.

Figure 7-32 shows performance of a read of a remote location for the case study machines[Kri*96]. The bars on the left show the total read time, broken down into the overhead associated with issuing the read and the latency of the remaining communication and remote pro-

cessing. The bars on the right show the minimum time between reads in the steady state. For the CM-5, there is no opportunity for optimization, since the remote processor must handle the network transaction. In the Paragon, the remote message processor performs the read request and replies. The remote processing time is significant, because the message processor must read the message from the NI, service it, and write a response to the NI. Moreover, the message processor must perform a protection check and the virtual to physical translation. On the Paragon with OSF1/AD, the message processor runs in kernel mode and operates on physical addresses; thus, it performs the page table lookup for the requested address, which may not be for the currently running process, in software. The Meiko CS-2 provides read and write network transactions, where the source-to-destination circuit in the network is held open until the read response or write acknowledgment is returned. The remote processing is dedicated and uses a hardware page table to perform the virtual-to-physical transaction on the remote node. Thus, the read latency is considerably less than a pair of messages, but still substantial. If the remote message processor performs the read operation, the latency increases by an additional 8 μs. The NOW system achieves a small performance advantage by avoiding use of the remote processor. As in the Paragon, the message processor must perform the protection check and address translation, which are quite slow in the embedded message processor. In addition, accessing remote memory from the network interface involves a DMA operation and is costly. The major advantage of dedicated processing of remote memory operations in all of these systems is that the performance of the operation does not depend on the remote compute processor servicing incoming requests from the network.
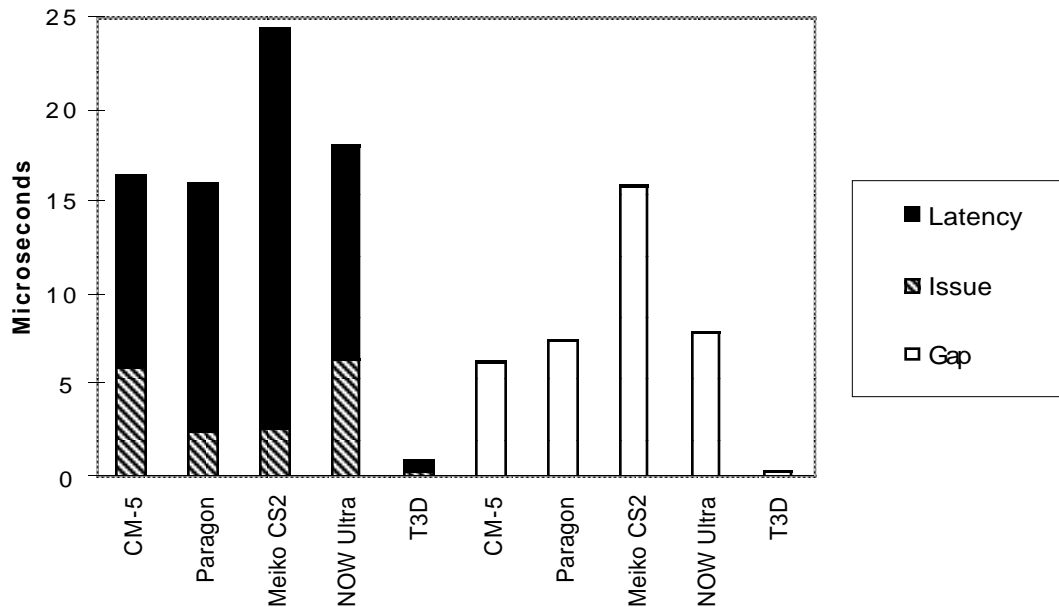
Figure 7-32 Performance Comparison on Shared Address Read

For the five study platforms the bars are the right show the total time to perform a remote read operation and isolate the portion of that time that involves the processor issuing and completing the read. The remainder can be overlapped with useful work, as is discussed in depth in Chapter 11. The bars on the right show the minimum time between successive reads, which is the reciprocal of the maximum rate.

The T3D shows an order of magnitude improvement available through dedicated hardware support for a shared address space. Given that the reads and writes are implemented through add-on hardware, there is a cost of 400 ns to issue the operation. The transmission of the request, remote service, and transmission of the reply take an additional 450 ns. For a sequence of reads, performance can be improved further by utilizing the hardware prefetch queue. Issuing the prefetch instruction and the pop from the queue takes about 200 ns. The latency is amortized over a sequence of such prefetches, and is almost completely hidden if 8 or more are issued.

7.9.3 Message Passing Operations

Let us also take a look at measured message passing performance of several large scale machines. As discussed earlier, the most common performance model for message passing operations is a linear model for the overall time to send n bytes given by

$$T(n) = T_0 + \frac{n}{B} \ .$$

(EQ 7.2)

The start-up cost, T_0, is logically the time to send zero bytes and B is the asymptotic bandwidth. The delivered data bandwidth is simply $BW(n) = \frac{n}{T(n)}$. Equivalently, the transfer time can be characterized by two parameters, r_∞ and $n_{\frac{1}{2}}$, which are the asymptotic bandwidth and the transfer size at which half of this bandwidth is obtained, *i.e.,* the half-power point.

The start-up cost reflects the time to carry out the protocol, as well as whatever buffer management and matching required to set up the data transfer. The asymptotic bandwidth reflects the rate at which data can be pumped through the system from end to end.

While this model is easy to understand and useful as a programming guide, it presents a couple of methodological difficulties for architectural evaluation. As with network transactions, the total message time is difficult to measure, unless a global clock is available, since the send is performed on one processor and the receive on another. This problem is commonly avoided by measuring the time for an echo test – one processor sends the data and then waits to receive it back. However, this approach only yields a reliable measurement if the receive is posted before the message arrives, for otherwise it is measuring the time for the remote node to get around to issuing the receive. If the receive is preposted, then the test does not measure the full time of the receive and it does not reflect the costs of buffering data, since the match succeeds. Finally, the measured times are not linear in the message size, so fitting a line to the data yields a parameter that has little to do with the actual startup cost. Usually there is a flat region for small values of n, so the start-up cost obtained through the fit will be smaller than the time for a zero byte message, perhaps even negative. These methodological concerns are not a problem for older machines, which had very large start-up costs and simple, software based message passing implementations.

Table 7-1 shows the start-up cost and asymptotic bandwidth reported for commercial message passing libraries on several important large parallel machines over a period of time[Don96]. (Also shown in the table are two small-scale SMPs, discussed in the previous chapter.) While the start-up cost has dropped by an order of magnitude in less than a decade, this improvement essentially tracks improvements in cycle time, as indicated by the middle column of the table. As illus-

trated by the right hand columns, improvements in start up cast have failed to keep pace with improvements in either floating-point performance or in bandwidth. Notice that the start-up costs are on an entirely different scale from the hardware latency of the communication network, which is typically a few microseconds to a fraction of a microsecond. It is dominated by processing overhead on each message transaction, the protocol and the processing required to provide the synchronization semantics of the message passing abstraction.

Table 7-1 Message passing start-up costs and asymptotic bandwidths

	Year	T_0 (μs)	max BW (MB/s)	Cycles per T_0	MFLOPS per proc	fp ops per T_0	$n_{\frac{1}{2}}$
iPSC/2	88	700	2.7	5600	1	700	1400
nCUBE/2	90	160	2	3000	2	300	300
iPSC/860	91	160	2	6400	40	6400	320
CM5	92	95	8	3135	20	1900	760
SP-1	93	50	25	2500	100	5000	1250
Meiko CS2	93	83	43	7470	24 (97)[a]	1992 (8148)	3560
Paragon	94	30	175	1500	50	1500	7240
SP-2	94	35	40	3960	200	7000	2400
Cray T3D (PVM)	94	21	27	3150	94	1974	1502
NOW	96	16	38	2672	180	2880	4200
SGI Power Challenge	95	10	64	900	308	3080	800
Sun E5000	96	11	160	1760	180	1980	2100

a. Using Fujitsu vector units

Figure 7-33 shows the measured one way communication time, on the echo test, for several machines as a function of message size. In this log-log plot, the difference in start-up costs is apparent, as is the non-linearity for small messages. The bandwidth is given by the slope of the lines. This is more clearly seen from plotting the equivalent $BW(n)$ in Figure 7-34. A caveat must be made in interpreting the bandwidth data for message passing as well, since the pairwise bandwidth only reflects the data rate on a point-to-point basis. We know, for example, that for the bus-based machines this is not likely to be sustained if many pairs of nodes are communicating. We will see in Chapter 7 that some network can sustain much higher aggregate bandwidths than others. In particular, the Paragon data is optimistic for many communication patterns involving multiple pairs of nodes, whereas the other large scale systems can sustain the pairwise bandwidth for most patterns.

7.9.4 Application Level Performance

The end-to-end effects of all aspects of the computer system come together to determine the performance obtained at the application level. This level of analysis is most useful to the end user, and is usually the basis for procurement decisions. It is also the ultimate basis for evaluating architectural trade-offs, but this requires mapping cumulative performance effects down to root
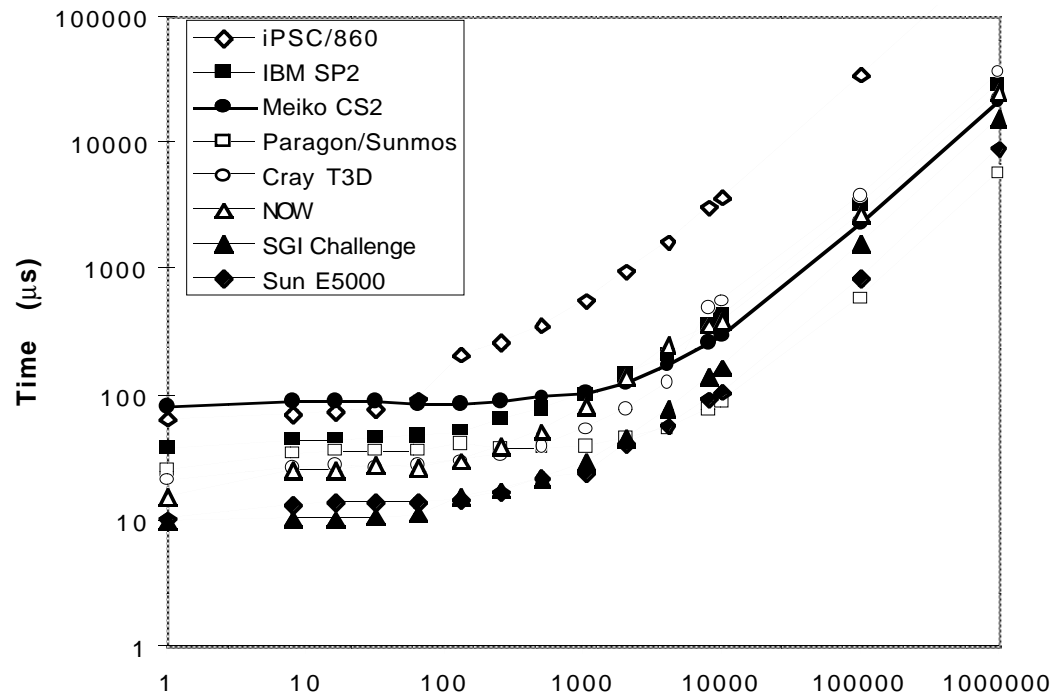
Figure 7-33 Time for Message Passing Operation vs. Message Size

Message passing implementations exhibit nearly an order of magnitude spread in start-up cost and the cost is constant for a range of small message sizes, introducing a substantial nonlinearity in the time per message. The time is nearly linear for a range of large messages, where the slope of the curve gives the bandwidth.

causes in the machine and in the program. Typically, this involves profiling the application to isolate the portions where most time is spent and extracting the usage characteristics of the application to determine its requirements. This section briefly compares performance on our study machines for two the NAS parallel benchmarks in the NPB2 suite[NPB].

The LU benchmark solves a finite difference discretization of the 3-D compressible Navier-Stokes equations used for modeling fluid dynamics through a block-lower-triangular block-upper-triangular approximate factorization of the difference scheme. The LU factored form is cast as a relaxation, and is solved by Symmetric Successive Over Relaxation (SSOR). The LU benchmark is based on the NAS reference implementation from 1991 using the Intel message passing library, NX[Bar*93]. It requires a power-of-two number of processors. A 2-D partitioning of the 3-D data grid onto processors is obtained by halving the grid repeatedly in the first two dimensions, alternating between x and y, until all processors are assigned, resulting in vertical pencil-like grid partitions. Each pencil can be thought of as a stack of horizontal tiles. The ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively sweep from one corner on a given z plane to the opposite corner of the same z plane, thereupon proceeding to the next z plane. This constitutes a diagonal pipelining method and is called a ``wavefront'' method by its authors [Bar*93]. The software pipeline spends relatively lit-
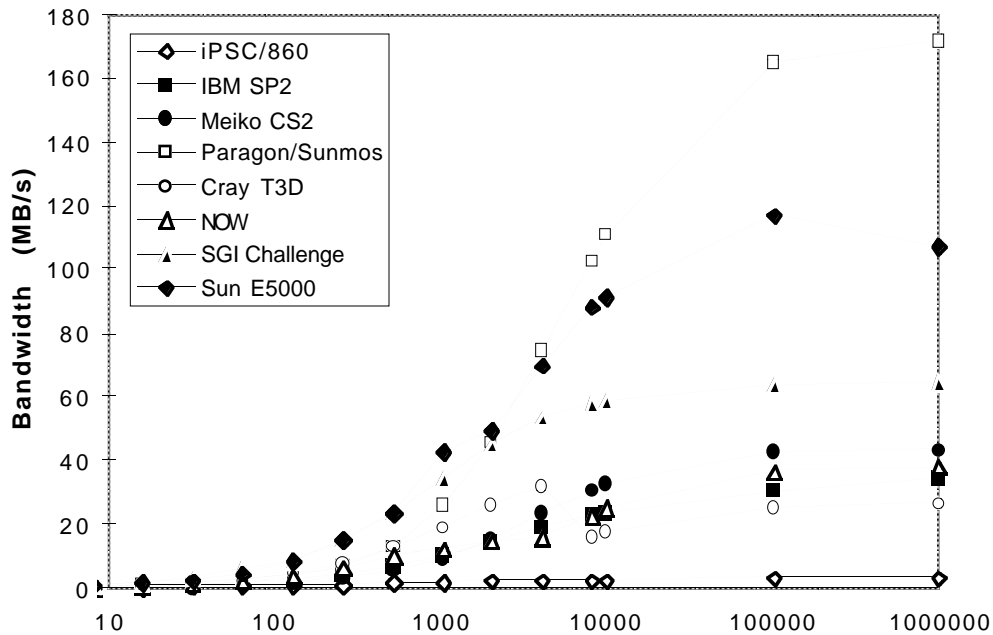
Figure 7-34 Bandwidth vs. Message Size

Scalable machines generally provide a few tens of MB/s per node on large message transfers, although the Paragon design shows that this can be pushed into the hundreds of MB/s. The bandwidth is limited primarily by the ability of the DMA support to move data through the memory system, including the internal busses. Small scale shared memory designs exhibit a point-to-point bandwidth dictated by out-of-cache memory copies when few simultaneous transfers are on-going.

tle time filling and emptying and is perfectly load-balanced. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition. It results in a relatively large number of small communications of 5 words each. Still, the total communication volume is small compared to computation expense, making this parallel LU scheme relatively efficient. Cache block reuse in the relaxation sections is high.

Figure 7-35 shows the speedups obtained on sparse LU with the Class A input (200 iterations on a 64x64x64 grid) for the IBM SP-2 (wide nodes), Cray T3D, and UltraSparc cluster (NOW). The speedup is normalized to the performance on four processors, shown in the right-hand portion of the figure, because this is the smallest number of nodes for which the problem can be run on the T3D. We see that Scalability is generally good to beyond a hundred processors, but both the T3D and NOW scale considerably better than the SP2. This is consistent with the ratio of the processor performance to the performance of small message transfers.

The BT algorithm solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. These systems are block tridiagonal with 5x5 blocks and are solved using a multi-partition scheme[Bru88]. The multi-partition approach provides good load balance and uses coarse grained communication. Each processor is responsible for several disjoint sub-blocks of points (``cells'') of the grid. The cells are arranged such that for each direction of the
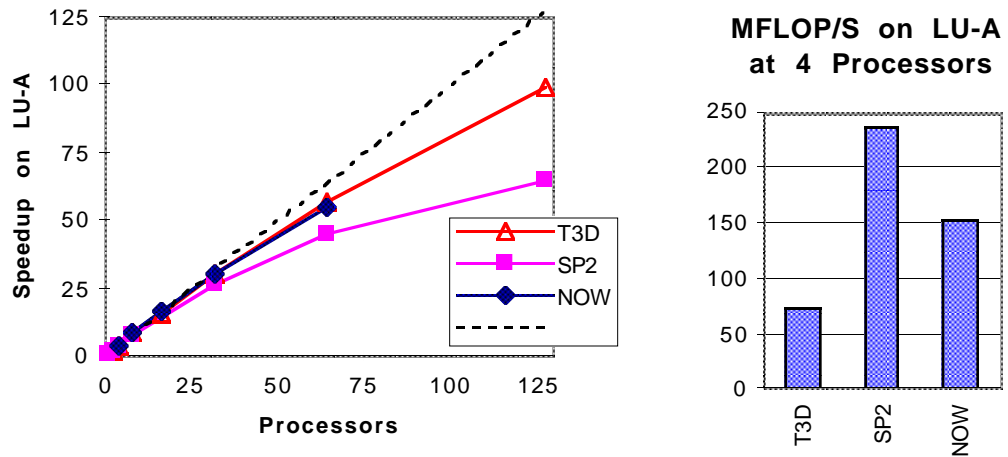
Figure 7-35 Application Performance on Sparse LU NAS Parallel Benchmark

Scalability of the IBM SP2, Cray T3D, and a UltraSparc cluster, on the sparse LU benchmark is shown normalized to the performacne obtained on four processors. The base performance of the three systems is shown on the right.
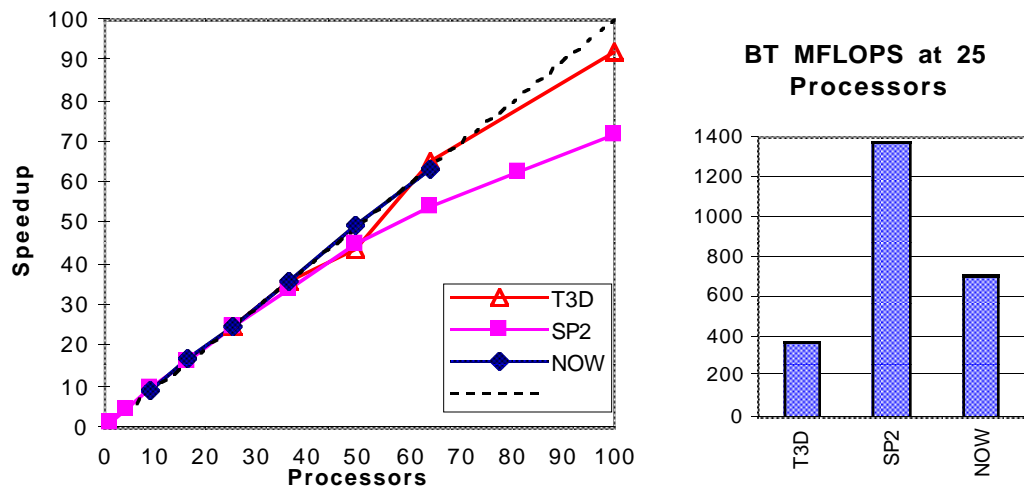
Figure 7-36 Application Performance on BT NAS Parallel Benchmark (in F77 and MPI)

Scalability of the IBM SP2, Cray T3D, and a UltraSparc cluster, on the BT benchmark is shown normalized to the performacne obtained on 25 processors. The base performance of the three systems is shown on the right.

line solve phase the cells belonging to a certain processor will be evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent. The BT codes require a

square number of processors. These codes have been written so that if a given parallel platform only permits a power-of-two number of processors to be assigned to a job, then unneeded processors are deemed inactive and are ignored during computation, but are counted when determining Mflop/s rates.

Figure 7-36 shows the scalability of the IBM SP2, Cray T3D, and the UltraSparc NOW on the Class A problem of the BT benchmark. Here the speedup is normalized to the performance on 25 processors, shown in the right-hand portion of the figure, because that is the smallest T3D configuration for which performance data is available. The scalability of all three platforms is good, with the SP-2 still lagging somewhat, but having much higher per node performance.

Table 7-2 Communication characteristics for one iteration of BT on class A problem over a range of processor count

	4 processors			16 processors			36 processors			64 processors		
	bins (KB)	msgs	KB	bins (KB)	msgs	KB	bins (KB)	msgs	KB	bins (KB)	msgs	KB
	43.5+	12	513	11.5+	144	1,652	4.5+	540	2,505	3+	1,344	4,266
	81.5+	24	1,916	61+	96	5,472	29+	540	15,425	19+	1,344	25,266
	261+	12	3,062	69+	144	9,738	45+	216	9,545	35.5+	384	13,406
Total KB			5,490			17,133			27,475			42,938
Time per iter			*5.43 s*			*1.46 s*			*0.67 s*			*0.38*
ave MB/s			1.0			11.5			40.0			110.3
ave MB/s/P			0.25			0.72			1.11			1.72

To understand why the scalability is less than perfect we need to look more closely at the characteristics of the application, in particular at the communication characteristics. To focus our attention on the dominant portion of the application we will study one iteration of the main outermost loop. Typically, the application performs a few hundred of these iterations, after an initialization phase. Rather than employ the simulation methodology of the previous chapters to examine communication characteristics, we collect this information by running the program using an instrumented version of the MPI message passing library. One useful characterization is the histogram of messages by size. For each message that is sent, a counter associated with its size bin is incremented. The result, summed over all processors, is shown in the top portion of Table 7-2 for a fixed problem size and processors scaled from four to 64. For each configuration, the non-zero bins are indicated along with the number of messages of that size and an estimate of the total amount of data transferred in those messages. We see that this application essentially sends three sizes of messages, but these sizes and frequencies vary strongly with the number of processors. Both of these properties are typical of message passing programs. Well-tuned programs tend to be highly structured and use communication sparingly. In addition, since the problem size is held fixed, as the number of processors increases the portion of the problem that each processor is responsible for decreases. Since the data transfer size is determined by how the program is coded, rather than by machine parameters, it can change dramatically with configuration. Whereas on small configurations the program sends a few very large messages, on a large configuration it sends many, relatively small messages. The total volume of communication increases by almost a

factor of eight when the number of processors increases by 16. This increase is certainly one factor affecting the speedup curve. In addition, the machine with higher start-up cost is affected more strongly by the decrease in message size.

It is important to observe that with a fixed problem size scaling rule, the workload on each processor changes with scale. Here we see it for communication, but it also changes cache behavior and other factors. The bottom portion of Table 7-2 gives an indication of the average communication requirement for this application. Taking the total communication volume and dividing by the time per iteration[1] on the Ultrasparc cluster we get the average delivered message data bandwidth. Indeed the communication rate scales substantially, increasing by a factor of more than one hundred over an increase in machine size of 16. Dividing further by the number of processors, we see that the average per processor bandwidth is significant, but not extremely large. It is in the same general ballpark as the rates we observed for shared address space applications on simulations of SMPs in Section 5.5. However, we must be extremely careful about making design decisions based on average communication requirements because communication tends to be very bursty. Often substantial periods of computation are punctuated by intense communication. For the BT application we can get an idea of the temporal communication behavior by taking snapshots of the message histogram at regular intervals. The result is shown in Figure 7-37 for several iterations on one of the 64 processors executing the program. For each sample interval, the bar shows the size of the largest message sent in that interval. The this application the communication profile is similar on all processors because it follows a bulk synchronous style with all processors alternating between local computation and phases of communication. The three sizes of messages are clearly visible, repeating in a regular pattern with the two smaller sizes more common than the larger one. Overall there is substantially more white space than dark, so the average communication bandwidth is more of an indication of the ratio of communication to computation than it is the rate of communication during a communication phase.

7.10 Synchronization

Scalability is a primary concern in the combination of software and hardware that implements synchronization operations in large scale distributed memory machines. With a message passing programming model, mutual exclusion is a given since each process has exclusive access to its local address space. Point-to-point events are implicit in every message operation. The more interesting case is orchestrating global or group synchronization from point-to-point messages. An important issue here is balance: It is important that the communication pattern used to achieve the synchronization be balanced among nodes, in which case high message rates and efficient synchronization can be realized. In the extreme, we should avoid having all processes communicate with or wait for one process at a given time. Machine designers and implementors of message passing layers attempt to maximize the message rate in such circumstances, but only the program can relieve the load imbalance. Other issues for global synchronization are similar to those for a shared address space, discussed below.

---

1. The execution time is the only data in the table that is a property of the specific platform. All of the other communication characteristics are a property of the program and are the same when measured on any platform.
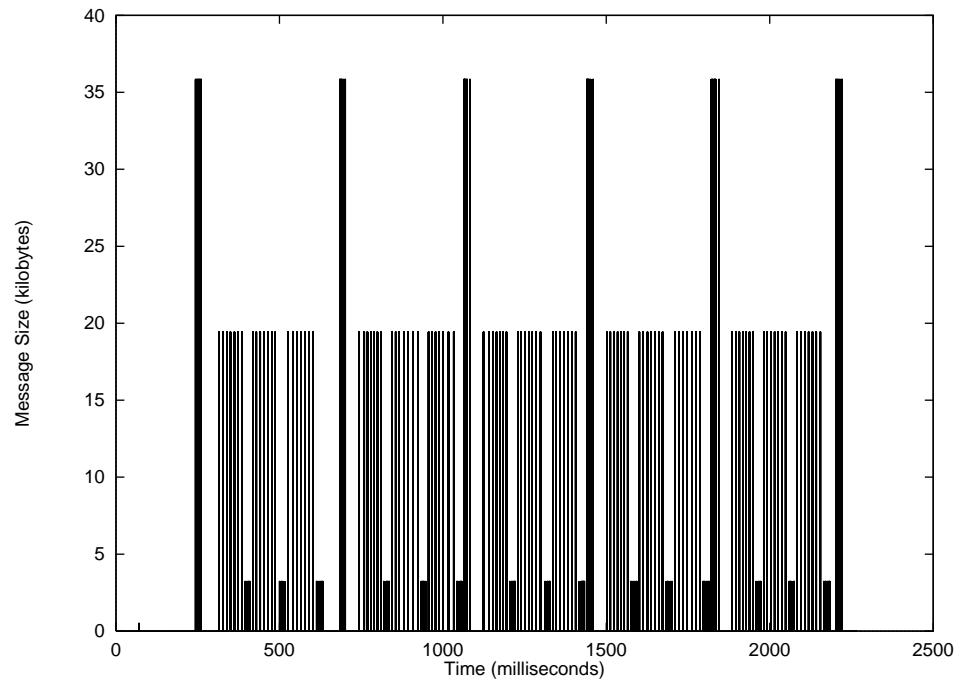
Figure 7-37 Message Profile over time for BT-A on 64 processors

The prgram is executed with an instrumented MPI library that samples the communication histogram at regular intervals. The graph shows the largest message sent from one particular processor during each sample interval. It is clear that communication is regular and bursty.

In a shared address space, the issues for mutual exclusion and point-to-point events are essentially the same as those discussed in the previous chapter. As in small-scale shared memory machines, the trend in scalable machines is to build user-level synchronization operations (like locks and barriers) in software on top of basic atomic exchange primitives. Two major differences, however, may affect the choice of algorithms. First, the interconnection network is not centralized but has many parallel paths. On one hand, this means that disjoint sets of processors can coordinate with one another in parallel on entirely disjoint paths; on the other hand, it can complicate the implementation of synchronization primitives. Second, physically distributed memory may make it important to allocate synchronization variables appropriately among memories. The importance of this depends on whether the machine caches nonlocal shared data or not, and is clearly greater for machines that don't, such as the ones described in this chapter. This section covers new algorithms for locks and barriers appropriate for machines with physically distributed memory and interconnect, starting from the algorithms discussed for centralized-memory machines in Chapter 5 (Section 5.6). Once we have studied scalable cache-coherent systems in the next chapter, we will compare the performance implications of the different algorithms for those machines with what we saw for machines with centralized memory in Chapter 5, and also examine the new issues raised for implementing atomic primitives. Let us begin with algorithms for locks.

7.10.1 Algorithms for Locks

In Section 5.6, we discussed the basic test&set lock, the test&set lock with back-off, the test-and-test&set lock, the ticket lock, and the array-based lock. Each successively went a step further in reducing bus traffic and fairness, but often at a cost in overhead. For example, the ticket lock allowed only one process to issue a test&set when a lock was released, but all processors were notified of the release through an invalidation and a subsequent read miss to determine who should issue the test&set. The array-based lock fixed this problem by having each process wait on a different location, and the releasing process notify only one process of the release by writing the corresponding location.

However, the array-based lock has two potential problems for scalable machines with physically distributed memory. First, each lock requires space proportional to the number of processors. Second, and more important for machines that do not cache remote data, there is no way to know ahead of time which location a process will spin on, since this is determined at runtime through a fetch&increment operation. This makes it impossible to allocate the synchronization variables in such a way that the variable a process spins on is always in its local memory (in fact, all of the locks in Chapter 5 have this problem). On a distributed-memory machine without coherent caches, such as the Cray T3D and T3E, this is a big problem since processes will spin on remote locations, causing inordinate amounts of traffic and contention. Fortunately, there is a software lock algorithm that both reduces the space requirements and also ensures all spinning will be on locally allocated variables. We describe this algorithm next.

Software Queuing Lock

This lock is a software implementation of a lock originally proposed for all-hardware implementation by the Wisconsin Multicube project [Goo89]. The idea is to have a distributed linked list or a queue of waiters on the lock. The head node in the list represents the process that holds the lock. Every other node is a process that is waiting on the lock, and is allocated in that process's local memory. A node points to the process (node) that tried to acquire the lock just after it. There is also a tail pointer which points to the last node in the queue, i.e. the last node to have tried to acquire the lock. Let us look pictorially at how the queue changes as processes acquire and release the lock, and then examine the code for the acquire and release methods.

Assume that the lock in Figure 7-38 is initially free. When process A tries to acquire the lock, it gets it and the queue looks as shown in Figure 7-38(a). In step (b), process B tries to acquire the lock, so it is put on the queue and the tail pointer now points to it. Process C is treated similarly when it tries to acquire the lock in step (c). B and C are now spinning on local flags associated with their queue nodes while A holds the lock. In step (d), process A releases the lock. It then "wakes up" the next process, B, in the queue by writing the flag associated with B's node, and leaves the queue. B now holds the lock, and is at the head of the queue. The tail pointer does not change. In step (e), B releases the lock similarly, passing it on to C. There are no other waiting processes, so C is at both the head and tail of the queue. If C releases the lock before another process tries to acquire it, then the lock pointer will be NULL and the lock will be free again. In this way, processes are granted the lock in FIFO order with regard to the order in which they tried to acquire it. The latter order will be defined next.

The code for the acquire and release methods is shown in Figure 7-39. In terms of primitives needed, the key is to ensure that changes to the tail pointer are atomic. In the acquire method, the
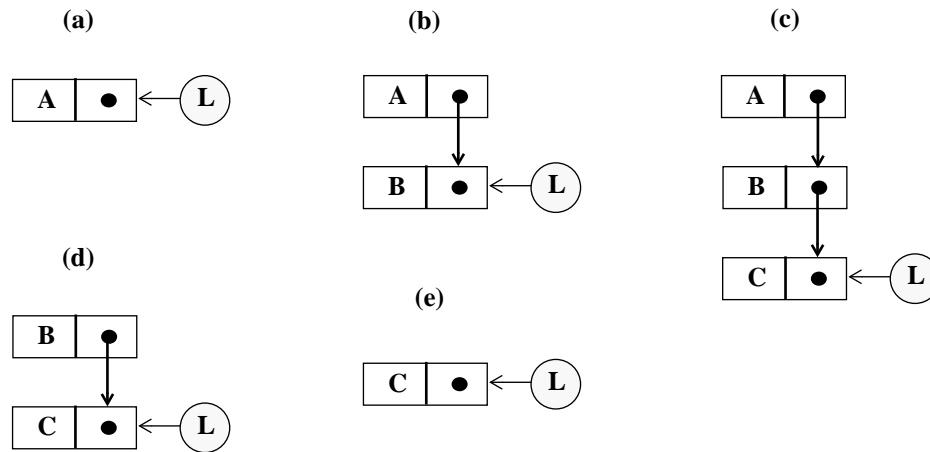
Figure 7-38 States of the queue for a lock as processes try to acquire and as processes release.

acquiring process wants to change the lock pointer to point to its node. It does this using an atomic *fetch&store* operation, which takes two operands: It returns the current value of the first operand (here the current tail pointer), and then sets it to the value of the second operand, returning only when it succeeds. The order in which in which the atomic fetch&store operations of different processes succeed defines the order in which they acquire the lock.

In the release method, we want to atomically check if the process doing the release is the last one in the queue, and if so set the lock pointer to NULL. We can do this using an atomic *compare&swap* operation, which takes three operands: It compares the first two (here the tail pointer and the node pointer of the releasing process), and if they are equal it sets the first (the tail pointer) to the third operand (here NULL) and returns TRUE; if they are not equal it does nothing and returns FALSE. The setting of the lock pointer to NULL must be atomic with the comparison, since otherwise another process could slip in between and add itself to the queue, in which case setting the lock pointer to NULL would be the wrong thing to do. Recall from Chapter 5 that a compare&swap difficult to implement as a single machine instruction, since it requires three operands in a memory instruction (the functionality can, however, be implemented using load-locked and store-conditional instructions). It is possible to implement this queuing lock without a compare&swap—using only a fetch&store—but the implementation is a lot more complicated (it allows the queue to be broken and then repairs it) and it loses the FIFO property of lock granting [MiSc96].

It should be clear that the software queueing lock needs only as much space per lock as the number of processes waiting on or participating in the lock, not space proportional to the number of processes in the program. It is the lock of choice for machines that support a shared address space with distributed memory but without coherent caching[Ala97].

```
struct node {
     struct node *next;
     int locked;
} *mynode, *prev_node;
shared struct node *Lock;


lock (Lock, mynode) {
     mynode->next = NULL;    /* make me last on queue */
     prev_node = fetch&store(Lock, mynode);
          /*Lock currently points to the previous tail of the queue; atomically set prev_node to
           the Lock pointer and set Lock to point to my node so I am last in the queue */
     if (prev_node != NULL) {/* if by the time I get on the queue I am not the only one,
                                   i.e. some other process on queue still holds the lock */
          mynode->locked = TRUE;        /* Lock is locked by other process*/
          prev_node->next = mynode;  /* connect me to queue */
          while (mynode->locked) {};  /* busy-wait till I am granted the lock */
     }
}


unlock (Lock, mynode) {
     if (mynode->next == NULL) {        /* no one to release, it seems*/
          if compare&swap(Lock,mynode,NULL) /* really no one to release, i.e.*/
               return;         /* i.e. Lock points to me, then set Lock to NULL and return */
          while (mynode->next == NULL);/* if I get here, someone just got on
                    the queue and made my c&s fail, so I should wait till they set my next pointer to
                    point to them before I grant them the lock */
     }
     mynode->next->locked = FALSE; /*someone to release; release them */
}
```

Figure 7-39 Algorithms for software queueing lock.

7.10.2 Algorithms for Barriers

In both message passing and shared address space models, global events like barriers are a key concern. A question of considerable debate is whether special hardware support is needed for global operations, or whether sophisticated software algorithms upon point-to-point operations are sufficient. The CM-5 represented one end of the spectrum, with a special "control" network

providing barriers, reductions, broadcasts and other global operations over a subtree of the machine. Cray T3D provided hardware support for barriers, but the T3E does not. Since it is easy to construct barriers that spin only on local variables, most scalable machines provide no special support for barriers at all but build them in software libraries using atomic exchange primitives or LL/SC. Implementing these primitives becomes more interesting when has to interact with coherent caching as well, so we will postpone the discussion of implementation till the next chapter.

In the centralized barrier used on bus-based machines, all processors used the same lock to increment the same counter when they signalled their arrival, and all waited on the same flag variable until they were released. On a large machine, the need for all processors to access the same lock and read and write the same variables can lead to a lot of traffic and contention. Again, this is particularly true of machines that are not cache-coherent, where the variable quickly becomes a hot-spot as several processors spin on it without caching it.

It is possible to implement the arrival and departure in a more distributed way, in which not all processes have to access the same variable or lock. The coordination of arrival or release can be performed in phases or rounds, with subsets of processes coordinating with one another in each round, such that after a few rounds all processes are synchronized. The coordination of different subsets can proceed in parallel with no serialization needed across them. In a bus-based machine, distributing the necessary coordination actions wouldn't matter much since the bus serializes all actions that require communication anyway; however, it can be very important in machines with distributed memory and interconnect where different subsets can coordinate in different parts of the network. Let us examine a few such distributed barrier algorithms.

Software Combining Trees

A simple distributed way to coordinate the arrival or release is through a tree structure (see

Flat
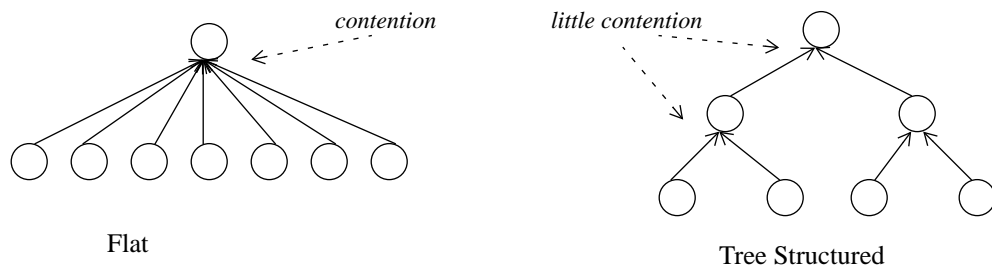
Tree Structured

Figure 7-40 Replacing a flat arrival structure for a barrier by an arrival tree (here of degree 2).

Figure 7-40), just as we suggested for avoiding hot-spots in Chapter 3. An arrival tree is a tree that processors use to signal their arrival at a barrier. It replaces the single lock and counter of the centralized barrier by a tree of counters. The tree may be of any chosen degree or branching factor, say k. In the simplest case, each leaf of the tree is a process that participate in the barrier. When a process arrives at the barrier, it signals its arrival by performing a fetch&increment on the counter associated with its parent. It then checks the value returned by the fetch&increment to see if it was the last of its siblings to arrive. If not, it's work for the arrival is done and it simply waits for the release. If so, it considers itself chosen to represent its siblings at the next level of

the tree, and so does a fetch&increment on the counter at that level. In this way, each tree node sends only a single representative process up to the next higher level in the tree when all the processes represented by that node's children have arrived. For a tree of degree k, it takes $\log_k p$ levels and hence that many steps to complete the arrival notification of p processes. If subtrees of processes are placed in different parts of the network, and if the counter variables at the tree nodes are distributed appropriately across memories, fetch&increment operations on nodes that do not have an ancestor-descendent relationship need not be serialized at all.

A similar tree structure can be used for the release as well, so all processors don't busy-wait on the same flag. That is, the last process to arrive at the barrier sets the release flag associated with the root of the tree, on which only k-1 processes are busy-waiting. Each of the k processes then sets a release flag at the next level of the tree on which k-1 other processes are waiting, and so on down the tree until all processes are released. The critical path length of the barrier in terms of number of dependent or serialized operations (e.g. network transactions) is thus $O(log_k\ p)$, as opposed to $O(p)$ for the centralized barrier or $O(p)$ for any barrier on a centralized bus. The code for a simple combining tree barrier tree is shown in Figure 7-41.

While this tree barrier distributes traffic in the interconnect, for machines that do not cache remote shared data it has the same problem as the simple lock: the variables that processors spin on are not necessarily allocated in their local memory. Multiple processors spin on the same variable, and which processors reach the higher levels of the tree and spin on the variables there depends on the order in which processors reach the barrier and perform their fetch&increment instructions, which is impossible to predict. This leads to a lot of network traffic while spinning.

Tree Barriers with Local Spinning

There are two ways to ensure that a processor spins on a local variable. One is to predetermine which processor moves up from a node to its parent in the tree based on process identifier and the number of processes participating in the barrier. In this case, a binary tree makes local spinning easy, since the flag to spin on can be allocated in the local memory of the spinning processor rather than the one that goes up to the parent level. In fact, in this case it is possible to perform the barrier without any atomic operations like fetch&increment, but with only simple reads and writes as follows. For arrival, one process arriving at each node simply spins on an arrival flag associated with that node. The other process associated with that node simply writes the flag when it arrives. The process whose role was to spin now simply spins on the release flag associated with that node, while the other process now proceeds up to the parent node. Such a static binary tree barrier has been called a "tournament barrier" in the literature, since one process can be thought of as dropping out of the tournament at each step in the arrival tree. As an exercise, think about how you might modify this scheme to handle the case where the number of participating processes is not a power of two, and to use a non-binary tree.

The other way to ensure local spinning is to use p-node trees to implement a barrier among p processes, where each tree node (leaf or internal) is assigned to a unique process. The arrival and wakeup trees can be the same, or they can be maintained as different trees with different branching factors. Each internal node (process) in the tree maintains an array of arrival flags, with one entry per child, allocated in that node's local memory. When a process arrives at the barrier, if its tree node is not a leaf then it first checks its arrival flag array and waits till all its children have signalled their arrival by setting the corresponding array entries. Then, it sets its entry in its parent's (remote) arrival flag array and busy-waits on the release flag associated with its tree node in

```
struct tree_node {
     int count = 0;                    /* counter initialized to 0 */
     int local_sense;                  /* release flag implementing sense reversal */
     struct tree_node *parent;
}


struct tree_node tree[P];    /* each element (node) allocated in a different memory */
private int sense = 1;
private struct tree_node *myleaf;/* pointer to this process's leaf in the tree */


barrier () {
     barrier_helper(myleaf);
     sense = !(sense);            /* reverse sense for next barrier call */
}


barrier_helper(struct tree_node *mynode) {
     if (fetch&increment (mynode->count) == k-1){/* last to reach node */
         if (mynode->parent != NULL)
               barrier_helper(mynode->parent); /*go up to parent node */
          mynode->count = 0;                    /*set up for next time */
          mynode->local_sense = !(mynode->local_sense);/* release */
     endif
     while (sense != mynode->local_sense) {};/* busy-wait */
}
```

Figure 7-41 A software combining barrier algorithm, with sense-reversal.

the wakeup tree. When the root process arrives and when all its arrival flag array entries are set, this means that all processes have arrived. The root then sets the (remote) release flags of all its children in the wakeup tree; these processes break out of their busy-wait loop and set the release flags of their children, and so on until all processes are released. The code for this barrier is shown in Figure 7-42, assuming an arrival tree of branching factor 4 and a wakeup tree of branching factor 2. In general, choosing branching factors in tree-based barriers is largely a trade-off between contention and critical path length counted in network transactions. Either of these types of barriers may work well for scalable machines without coherent caching.

Parallel Prefix

In many parallel applications a point of global synchronization is associated with combining information that has been computed by many processors and distributing a result based on the

```
struct tree_node {
     struct tree_node *parent;
     int parent_sense = 0;
     int wkup_child_flags[2];      /* flags for children in wakeup tree */
     int child_ready[4];           /* flags for children in arrival tree */
     int child_exists[4];
}
/* nodes are numbered from 0 to P-1 level-by-level starting from the root.
struct tree_node tree[P];   /* each element (node) allocated in a different memory */
private int sense = 1, myid;
private me = tree[myid];


barrier() {
     while (me.child_ready is not all TRUE) {};/* busy-wait */
     set me.child_ready to me.child_exists;  /* re-initialize for next barrier call */
     if (myid !=0) {                /*set parent's child_ready flag, and wait for release */
          tree[⌊myid−1/4⌋].child_ready[(myid-1) mod 4] = true;
          while (me.parent_sense != sense) {};
     }
     me.child_pointers[0] = me.child_pointers[1] = sense;
     sense = !sense;
}
```

Figure 7-42 A combining tree barrier that spins on local variables only.

combination. Parallel prefix operations are an important, widely applicable generalization of reductions and broadcasts[Ble93]. Given some associative, binary operator $\oplus$ we want to compute $S_i = x_i \oplus x_{i-1} \ldots \oplus x_0$ for $i = 0, \ldots, P$. A canonical example is a running sum, but several other operators are useful. The carry-look-ahead operator from adder design is actually a special case of a parallel prefix circuit. The surprising fact about parallel prefix operations is that they can be performed as quickly as a reduction followed by a broadcast, with a simple pass up a binary tree and back down. Figure 7-43 shows the upward sweep, in which each node applies the operator to the pair of values it receives from its children and passes the result to its parent, just as with a binary reduction. (The value that is transmitted is indicated by the range of indices next to each arc; this is the subsequence over which the operator is applied to get that value.) In addition, each node holds onto the value it received from its least significant child (rightmost in the figure). Figure 7-44 shows the downward sweep. Each node waits until it receives a value from its parent. It passes this value along unchanged to its rightmost child. It combines this value with the value the was held over from the upward pass and passes the result to its left child. The nodes along the right edge of the tree are special, because they do not need to receive anything from their parent.

This parallel prefix tree can be implemented either in hardware or in software. The basic algorithm is important to understand.
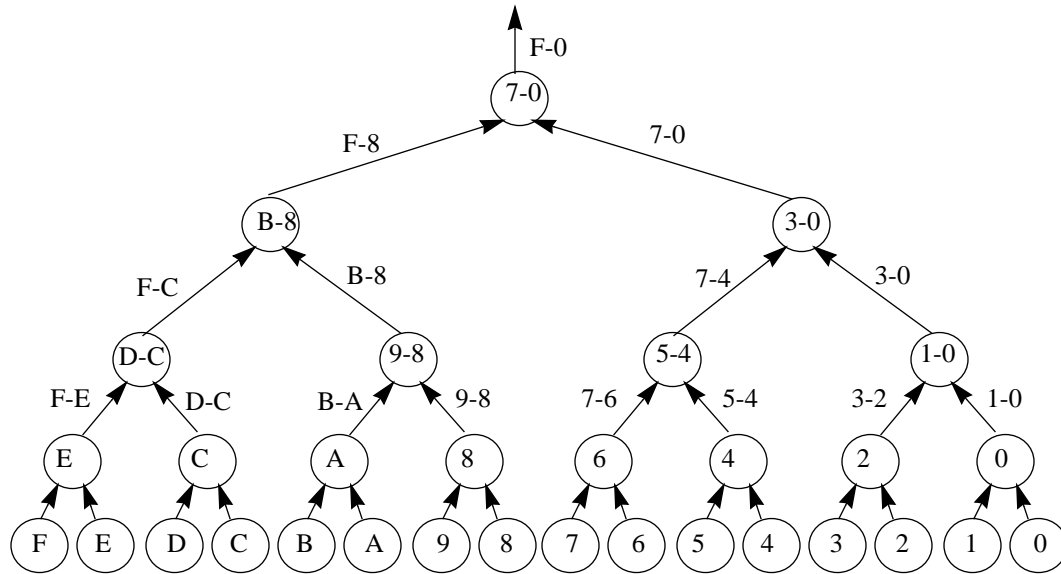
Figure 7-43 Upward sweep of the Parallel Prefix operation

Each node receive two elements from its children, combines them and passes thge result to its parent, and holds the element from the least significant (right) child
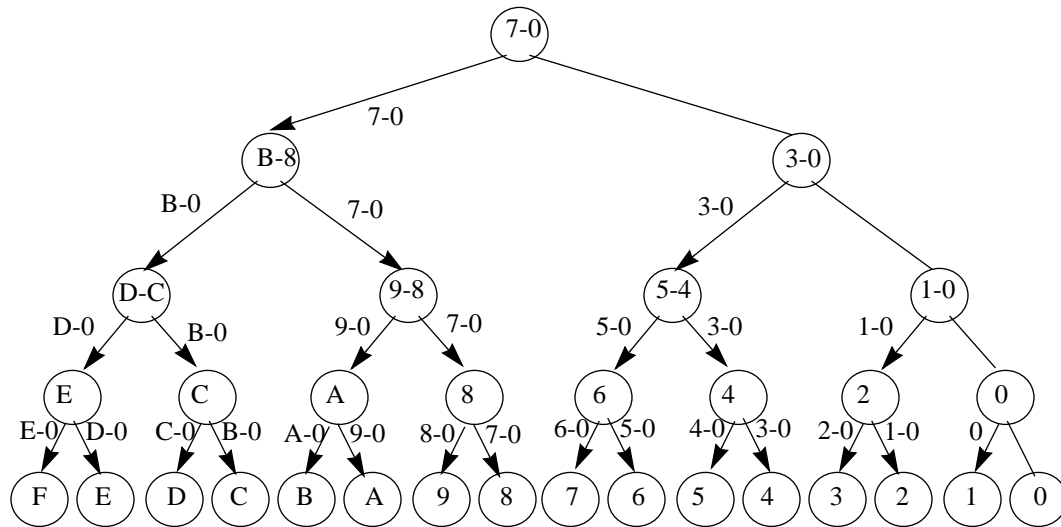
Figure 7-44 Downward sweep of the Parallel Prefix operation

When a node receives an element from above, it passes the data down to its right child, combines it with its stored element, and passes the result to its left child. Nodes along the rightmost branch need nothing from above.

All-to-all Personalized Communication

All-to-all personalized communication is when each process has a distinct set of data to transmit to every other process. The canonical example of this is a transpose operation, say, where each process owns a set of rows of a matrix and needs to transmit a sequence of elements in each row to every other processor. Another important example is remapping a data structure between blocked cyclic layouts. Many other permutations of this form are widely used in practice. Quite a bit of work has been done in implementing all-to-all personalized communication operations efficiently, i.e., with no contention internal to the network, on specific network topologies. If the network is highly scalable then the internal communication flows within the network become secondary, but regardless of the quality of the network contention at the endpoints of the network, i.e., the processors or memories is critical. A simple, widely used scheme is to schedule the sequence of communication events so that P rounds of disjoint pairwise exchanges are performed. In round i, process p transmits the data it has for process $q = p \otimes i$ obtained as the exclusive-OR of the binary number for p and the binary representation of i. Since, exclusive-OR is commutative, $p = q \otimes i$, and the round is, indeed, an exchange.

7.11 Concluding Remarks

We have seen in this chapter that most modern large scale machines constructed from general purpose nodes with a complete local memory hierarchy augmented by a communication assist interfacing to a scalable network. However, there is a wide range of design options for the communication assist. The design is influenced very strongly by where the communication assist interfaces to the node architecture: at the processor, at the cache controller, at the memory bus, or at the I/O bus. It is also strongly influenced by the target communication architecture and programming model. Programming models are implemented on large scale machines in terms of protocols constructed out of primitive network transactions. The challenges in implemented such a protocol are that a large number of transactions can be outstanding simultaneously and there is no global arbitration. Essentially any programming model can be implemented on any of the available primitives at the hardware/software boundary, and many of the correctness and scalability issues are the same, however the performance characteristics of the available hardware primitives are quite different. The performance ultimately influences how the machine is viewed by programmers.

Modern large scale parallel machine designs are rooted heavily in the technological revolution of the mid 80s – the single chip microprocessor - which was coined the "killer micro" as the MPP machines began to take over the high performance market from traditional vector supercomputers. However, these machines pioneered a technological revolution of their own - the single chip scalable network switch. Like the microprocessor, this technology has grown beyond its original intended use and a wide class of scalable system area networks are emerging, including switched gigabit (perhaps multi-gigabit) ethernets. As a result, large scale parallel machine design has split somewhat into two branches. Machines oriented largely around message passing concepts and explicit get/put access to remote memory are being overtaken by clusters, because of their extreme low cost, ease of engineering, and ability to track technology. The other branch is machines that deeply integrated the network into the memory system to provide cache coherent access to a global physical address space with automatic replication. In other words, machine that look to the programmer like those of the previous chapter, but are built like the machines in this

chapter. Of course, the challenge is advancing the cache coherence mechanisms in a manner that provides scalable bandwidth and low latency. This is the subject of the next chapter.

7.12 References

[Ala97]   Alain Kägi, Doug Burger, and James R. Goodman, Efficient Synchronization: Let Them Eat QOLB, Proc. of 24th International Symposium on Computer Architecture, June, 1997.

[Alp92]   Alpha Architecture Handbook, Digital Equipment Corporation, 1992.

[And*92b]   T. Anderson, S. Owicki, J. Saxe and C. Thacker, "High Speed Switch Scheduling for Local Area Networks", Proc. *Fifth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, October 1992.

[And*94]   Anderson, T.E.; Culler, D.E.; Patterson, D. A case for NOW (Networks of Workstations), IEEE Micro, Feb. 1995, vol.15, (no.1):54-6

[Arn*89]   E.A. Arnould, F.J. Bitz, Eric Cooper, H.T. Kung, R.D. Sansom, and P.A. Steenkiste, "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers", Proc. *Third Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[Arp*95]   A. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the Cray-T3D: A Compiler Perspective. To appear in *Proc. of ISCA*, June 1995.

[AtSe88]   William C. Athas and Charles L. Seitz, Multicomputers: Message-Passing Concurrent Computers, IEEE Computer, Aug. 1988, pp. 9-24.

[BaP93]   D. Banks and M. Prudence, "A High Performance Network Architecture for a PA-RISC Workstation", *IEEE J. on Selected Areas in Communication*, 11(2), Feb. 93.

[Bar*93]   Barszcz, E.; Fatoohi, R.; Venkatakrishnan, V.; and Weeratunga, S.: ``Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors'' Technical Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA, 94035-1000, April 1993.

[Bar*94]   E. Barton, J. Crownie, and M. McLaren. Message Passing on the Meiko CS-2. Parallel Computing, 20(4):497-507, Apr. 1994.

[Ble93]   Guy Blelloch, Prefix Sums and Their Applications. In Synthesis of Parallel Algorithms ed John Reif, Morgan Kaufmann, 1993, pp 35-60.

[Blu*94]   M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, M.R. Mesarina, "Two Virtual Memory Mapped Network Interface Designs", Hot Interconnects II Symposium, Aug. 1994.

[Bod*95]   N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su, Myrinet: A Gigabit-per-Second Local Area Network, IEEE Micro, Feb. 1995, vol.15, (no.1), pp. 29-38.

[BoRo89]   Luc Bomans and Dirk Roose, Benchmarking the iPSC/2 Hypercube Multiprocessor. Concurrency: Practivce and Experience, 1(1):3-18, Sep. 1989.

[Bor*90]   S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iwarp.In Proc. 17th Intl. Symposium on Computer Architecture, pages 70-81. ACM, May 1990. Revised version appears as technical report CMU-CS-90-197.

[BBN89]   BBN Advanced Computers Inc., TC2000 Technical Product Summary, 1989.

[Bru88]   Bruno, J; Cappello, P.R.: Implementing the Beam and Warming Method on the Hypercube. Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena,

CA, Jan 19-20, 1988.

[BCS93]   Edoardo Biagioni, Eric Cooper, and Robert Sansom, "Designing a Practical ATM LAN", *IEEE Network*, March 1993.

[Car*96] W. Cardoza, F. Glover, and W. Snaman Jr., "Design of a TruCluster Multicomputer System for the Digital UNIX Environment", Digital Technical Journal, vol. 8, no. 1, pp. 5-17, 1996.

[Cul*91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, Fine-grain Parallelism with Minimal Hardware Support, Proc. of the 4th Int'l Symposium on Arch. Support for Programming Languages and Systems (ASPLOS), pp. 164-175, Apr. 1991.

[DaSe87] W. Dally and C. Seitz, Deadlock-Free Message Routing in Multiprocessor Interconnections Networks, IEEE-TOC, vol c-36, no. 5, may 1987.

[Dal90]   William J. Dally, Performance Analysis of k-ary n-cube Interconnection Networks, IEEE-TOC, V. 39, No. 6, June 1990, pp. 775-85.

[Dal93]   Dally, W.J.; Keen, J.S.; Noakes, M.D., The J-Machine architecture and evaluation, Digest of Papers. COMPCON Spring '93, San Francisco, CA, USA, 22-26 Feb. 1993, p. 183-8.

[Dub96]   C. Dubnicki, L. Iftode, E. W. Felten, K. Li, "Software Support for Virtual Memory-Mapped Communication", 10th International Parallel Processing Symposium, April 1996.

[Dun88]   T. H. Dunigan, Performance of a Second Generation Hypercube, Technical Report ORNL/TM-10881, Oak Ridge Nat. Lab., Nov. 1988.

[Fox*88]G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems on Concurrent Processors, vol 1, Prentice-Hall, 1988.

[Gei93]   A. Geist et al, PVM 3.0 User's Guide and REference Manual, Oak Ridge NAtional Laboratory, Oak Ridge, Tenn. Tech. Rep. ORNL/TM-12187, Feb. 1993.

[Gil96]   Richard Gillett, Memory Channel Network for PCI, IEEE Micro, pp. 12-19, vol 16, No. 1., Feb. 1996.

[Gika97] Richard Gillett and Richard Kaufmann, Using Memory Channel Network, IEEE Micro, Vol. 17, No. 1, January / February 1997.

[Goo89]   James. R. Goodman, Mary. K. Vernon, Philip. J. Woest., Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor, Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1989.

[Got*83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, IEEE Trans. on Computers, c-32(2):175-89, Feb. 1983.

[Gro92]   W. Groscup, The Intel Paragon XP/S supercomputer. Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Nov 1992, pp. 262--273.

[GrHo90] Grafe, V.G.; Hoch, J.E., The Epsilon-2 hybrid dataflow architecture, COMPCON Spring '90, San Francisco, CA, USA, 26 Feb.-2 March 1990, p. 88-93.

[Gur*85]J. R. Gurd, C. C. Kerkham, and I. Watson, The Manchester Prototype Dataflow Computer, Communications of the ACM, 28(1):34-52, Jan. 1985.

[Hay*94] K. Hayashi, *et al*. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. *Proc. of ASPLOS VI*, October 1994.

[HoMc93] Mark Homewood and Moray McLaren, Meiko CS-2 Interconnect Elan - Elite Design, Hot Interconnects, Aug. 1993.

[Hor95]   R. Horst, TNet: A Reliable System Area Network, IEEE Micro, Feb. 1995, vol.15, (no.1), pp. 37-45.

[i860]   Intel Corporation. *i750, i860, i960 Processors and Related Products*, 1994.

[Kee*95]   Kimberly K. Keeton, Thomas E. Anderson, David A. Patterson, LogP Quantified: The Case for Low-Overhead Local Area Networks. Hot Interconnects III: A Symposium on High Performance Interconnects , 1995.

[KeSc93]   R. E. Kessler and J. L. Schwarzmeier, Cray T3D: a new dimension for Cray Research, Proc. of Papers. COMPCON Spring '93, pp 176-82, San Francisco, CA, Feb. 1993.

[Koe*94]   R. Kent Koeninger, Mark Furtney, and Martin Walker. A Shared Memory MPP from Cray Research, Digital Technical Journal 6(2):8-21, Spring 1994.

[Kri*96]   A. Krishnamurthy, K. Schauser, C. Scheiman, R. Wang, D. Culler, and K. Yelick, "Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines", Proc. of ASPLOS 96.

[Kro*86]   Kronenberg, Levy, Strecker "Vax Clusters: A closely-coupled distributed system," ACM Transactions on Computer Systems, May 1986 v 4(3):130-146.

[Kun*91]   Kung, H.T, et al, Network-based multicomputers: an emerging parallel architecture, Proceedings Supercomputing '91, Albuquerque, NM, USA, 18-22 Nov. 1991 p. 664-73

[Law*96]   J. V. Lawton, J. J. Brosnan, M. P. Doyle, S.D. O Riodain, T. G. Reddin, "Building a High-performance Message-passing System for MEMORY CHANNEL Clusters", Digital Technical Journal, Vol. 8 No. 2, pp. 96-116, 1996.

[Lei*92]   C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmau, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, R. Zak. The Network Architecture of the CM-5. Symposium on Parallel and Distributed Algorithms '92" Jun 1992, pp. 272-285.

[Lit*88]   M. Litzkow, M. Livny, and M. W. Mutka, ``Condor - A Hunter of Idle Workstations,'' Proceedings of the 8th International Conference of Distributed Computing Systems, pp. 104-111, June, 1988.

[LuPo97]   Jefferey Lukowsky and Stephen Polit, IP Packet Switching on the GIGAswitch/FDDI System, On-line as http://www.networks.digital.com:80/dr/techart/gsfip-mn.html.

[Mar*93]   Martin, R. "HPAM: An Active Message Layer of a Network of Workstations", presented at Hot Interconnects II, Aug. 1994.

[MCS91]   John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared Memory Mutiprocessors. ACM Transactions on Computer Systems, vol. 9, no. 1, pp. 21-65, February 1991.

[MiSc96]   M. Michael and M. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, pp 267-276, May 1996.

[MPI93]   MPI Forum, "MPI: A Message Passing Interface", *Supercomputing 93*, 1993. (Updated spec at http://www.mcs.anl.gov/mpi/)

[NiAr89]   R. S. Nikhil and Arvind, Can Dataflow Subsume von Neuman Computing?, Proc. of 16th Annual International Symp. on Computer Architecture, pp. 262-72, May 1989

[NPA93]   R. Nikhil, G. Papadopoulos, and Arvind, *T: A Multithreaded Massively Parallel Architecture, Proc. of ISCA 93, pp. 156-167, May 1993.

[NPB]   NAS Parallel Benchmarks, http://science.nas.nasa.gov/Software/NPB/.

[PaCu90]   Papadopoulos, G.M. and Culler, D.E. Monsoon: an explicit token-store architecture, Proceedings. The 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, 28-31 May 1990, p. 82-91.

[PiRe94]   Paul Pierce and Greg Regnier, The Paragon Implementation of the NX Message Passing Interface.

Proc. of the Scalable High-Performance Computing Conference, pp. 184-90, May 1994.

[Pfi*85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliff, E. A. Melton, V. A. Norton, and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, International Conference on Parallel Processing, 1985.

[Pfi95] G.F. Pfister, In Search of Clusters – the coming battle in lowly parallel computing, Prentice Hall, 1995.

[Rat85] J. Ratner, Concurrent Processing: a new direction in scientific computing, Conf. Proc. of the 1985 Natinal Computing Conference, p. 835, 1985.

[Sak*91] Sakai, S.; Kodama, Y.; Yamaguchi, Y., Prototype implementation of a highly parallel dataflow machine EM4. Proceedings. The Fifth International Parallel Processing Symposium, Anaheim, CA, USA, 30 April-2 May, p. 278-86.

[Sco96] Scott, S., Synchronization and Communication in the T3E Multiprocesor, Proc. of the 7th International Conference on Architectural Suport for Programming Languages and Operating Systems, Cambridge, MA, pp.. 26-36, Oct. 1996.

[Shi*84] T. Shimada, K. Hiraki, and K. Nishida, An Architecture of a Data Flow Machine and its Evaluation, Proc. of Compcon 84, pp. 486-90, IEEE, 1984.

[SP2] C. B. Stunkel *et al.* The SP2 Communication Subsystem. On-line as `http://ibm.tc.cornell.edu/ibm/pps/doc/css/css.ps`

[ScSc95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. IPPS '95.

[Sei84] C. L. Seitz, Concurrent VLSI Architectures, IEEE Transactions on Computers, 33(12):1247-65, Dec. 1984

[Sei85] Charles L. Seitz, The Cosmic Cube, Communications of the ACM, 28(1):22-33, Jan 1985.

[Swa*77a] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, CM* - a modular, multi-microprocessor, AFIPS Conference Proceedings, vol 46, National Computer Conference, 1977, pp. 637-44.

[Swa*77b] R. J. Swan, A. Bechtolsheim, K-W Lai, and J. K. Ousterhout, The Implementation of the CM* multi-microprocessor, AFIPS Conference Proceedings, vol 46, National Computer Conference, 1977, pp. 645-55.

[Tho64] Thornton, James E., Parallel Operation in the Control Data 6600, AFIPS proc. of the Fall Joint Computer Conference, pt. 2, vol. 26, pp. 33-40, 1964. (Reprinted in Computer Structures: Principles and Examples, Siework, Bell, and Newell, McGray Hill, 1982).

[TrS91] C. Traw and J. Smith, "A High-Performance Host Interface for ATM Networks," Proc. *ACM SIGCOMM Conference*, pp. 317-325, Sept. 1991.

[TrDu92] R. Traylor and D. Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. *Proc. of Hot Chips '92 Symposium*, August 1992.

[vEi*92] von Eicken, T.; Culler, D.E.; Goldstein, S.C.; Schauser, K.E., Active messages: a mechanism for integrated communication and computation, Proc. of 19th Annual International Symposium on Computer Architecture, old Coast, Qld., Australia, 19-21 May 1992, pp. 256-66

[vEi*95] T. von Eicken, A. Basu, and Vineet Buch, Low-Latency Communication Over ATM Using Active Messages, IEEE Micro, Feb. 1995, vol.15, (no.1), pp. 46-53.

[WoHi95] D. A. Wood and M. D. Hill, Cost-effective Parallel Computing, IEEE Computer, Feb. 1995, pp. 69-72.

7.13 Exercices

7.1 A radix-2 FFT over n complex numbers is implemented as a sequence of $\log n$ completely parallel steps, requiring $5n\log n$ floating point operations while reading and writing each element of data $\log n$ times. Calculate the communication to computation ratio on a dancehall design, where all processors access memory though the network, as in Figure 7-3. What communication bandwidth would the network need to sustain for the machine to deliver $250p$ MFLOPS on p processors?

7.2 If the data in Exercise 7.1 is spread over the memories in a NUMA design using either a cycle or block distribution, the $\log \frac{n}{p}$ of the steps will access data in the local memory, assuming both n and p are powers of two, and in the remaining $\log p$ steps half of the reads and half of the writes will be local. (The choice of hayout determines which steps are local and which are remote, but the ratio stays the same.) Calculate the communication to computation ratio on a distributed memory design, where each processor has a local memory, as in Figure 7-2. What communication bandwidth would the network need to sustain for the machine to deliver $250p$ MFLOPS on p processors? How does this compare with Exercise 7.1?

7.3 If the programmer pays attention to the layout, the FFT can be implemented so that a single, global transpose operation is required where $n - \frac{n}{p}$ data elements are trasmitted across the network. All of the $\log n$ steps are performed on local memory. Calculate the communication to computation ratio on a distributed memory design. What communication bandwidth would the network need to sustain for the machine to deliver $250p$ MFLOPS on p processors? How does this compare with Exercise 7.2?

7.4 Reconsider Example 7-1 where the number of hops for a n-node configuration is $\sqrt{n}$. How does the average transfer time increase with the number of nodes? What about $\sqrt[3]{n}$.

7.5 Formalize the cost scaling for the designs in Exercise 7.4.

7.6 Consider a machine as described in Example 7-1, where the number of links occupied by each transfer is $\log n$. In the absence of contention for individual links, how many transfers can occur simultaneously?

7.7 Reconsider Example 7-1 where the network is a simple ring. The average distance between two nodes on ring of n nodes is n/2. How does the average transfer time increase with the number of nodes? Assuming each link can be occupied by at most one transfer at a time, how many such transfers can take place simultaneously.

7.8 For a machine as described in Example 7-1, suppose that a broadcast from a node to all the other nodes use n links. How would you expect the number of simultaneous broadcasts to scale with the numebr of nodes?

7.9 Suppose a 16-way SMP lists at $10,000 plus $2,000 per node, where each node contains a fast processor and 128 MB of memory. How much does the cost increase when doubling the capcity of the system from 4 to 8 processors? From 8 to 16 processors?

7.10 Prove the statement from Section 7.2.3 that parallel computing is more cost-effective whenever $Speedup(p) > Costup(p)$.

7.11 Assume a bus transaction with n bytes of payload occupies the bus for $4 + \left\lceil \dfrac{n}{8} \right\rceil$ cycles, up to a limit of 64 bytes. Draw graph comparing the bus utilization for programmed I/O and DMA for sending various sized messages, where the message data is in memory, not in registers. For DMA include the extra work to inform the communication assist of the DMA address and

length. Consider both the case where the data is in cache and where it is not. What assumptions do you need to make about reading the status registers?

7.12 The Intel Paragon has an output buffer of size 2KB which can be filled from memory at a rate of 400 MB/s and drained into the network at a rate of 175MB/s. The buffer is drained into the network while it is being filled, but if the buffer gets full the DMA device will stall. In designing your message layer you decide to fragment long messages into DMA bursts that are as long as possible without stalling behind the output buffer. Clearly these can be at least 2KB is size, but in fact they can be longer. Calculate the apparent size of the output buffer driving a burst into an empty network given these rates of flow.

7.13 Based on a rough estimate from Figure 7-33, which of the machines will have a negative T_0 if a linear model is fit to the communication time data?

7.14 Use the message frequency data presented in Table 7-2 to estimate the time each processor would spend in communication on an iteration of BT for the machines described in Table 7-1.

7.15 Table 7-3 describes the communication characteristics of sparse LU.

- How does the message size characteristics differ from that of BT? What does this say about the application?

- How does the message frequency differ?

- Estimate the time each processor would spend in communication on an iteration of LU for the machines described in Table 7-1.

Table 7-3 Communication characteristics for one iteration of LU on class A problem over a range of processor count

	4 processors			16 processors			32 processors			64 processors		
	bins (KB)	msgs	KB	bins (KB)	msgs	KB	bins (KB)	msgs	KB	bins (KB)	msgs	KB
	1+	496	605	0.5-1	2,976	2,180	0-0.5	2,976	727	0-0.5	13,888	3,391
	163.5+	8	1,279	81.5+	48	3,382	0.5-1	3,472	2,543	81.5	224	8,914
							40.5+	48	1,910			
							81.5+	56	4,471			
Total KB												
Time per iter			2.4s			0.58			0.34			0.19
ave MB/s			0.77			10.1			27.7			63.2
ave MB/s/P			0.19			0.63			0.86			0.99

CHAPTER 8   Directory-based Cache Coherence

8.1 Introduction

A very important part of the evolution of parallel architecture puts together the concepts of the previous two chapters. The idea is to provide scalable machines supporting a shared physical address space, but with automatic replication of data in local caches. In Chapter 7 we studied how to build scalable machines by distributing the memory among the nodes. Nodes were connected together using a scalable network, a sophisticated communication assist formed the node-to-network interface as shown in Figure 8-1, and communication was performed through point-to-point network transactions. At the final point in our design spectrum, the communication assist provided a shared physical address space. On a cache miss, it was transparent to the cache and the processor whether the miss was serviced from the local physical memory or some remote memory; the latter just takes longer. The natural tendency of the cache in this design is to replicate the data accessed by the processor, regardless of whether the data are shared or from where they come. However, this again raises the issue of coherence. To avoid the coherence problem

and simplify memory consistency, the designs of Chapter 7 disabled the hardware caching of logically shared but physically remote data. This implied that the programmer had to be particularly careful about data distribution across physical memories, and replication and coherence were left to software. The question we turn to in this chapter is how to provide a shared address space programming model with transparent, coherent replication of shared data in the processor caches and an intuitive memory consistency model, just as in the small scale shared memory machines of Chapter 5 but without the benefit of a globally snoopable broadcast medium. Not only must the hardware latency and bandwidth scale well, as in the machines of the previous chapter, but so must the protocols used for coherence as well, at least up to the scales of practical interest. This
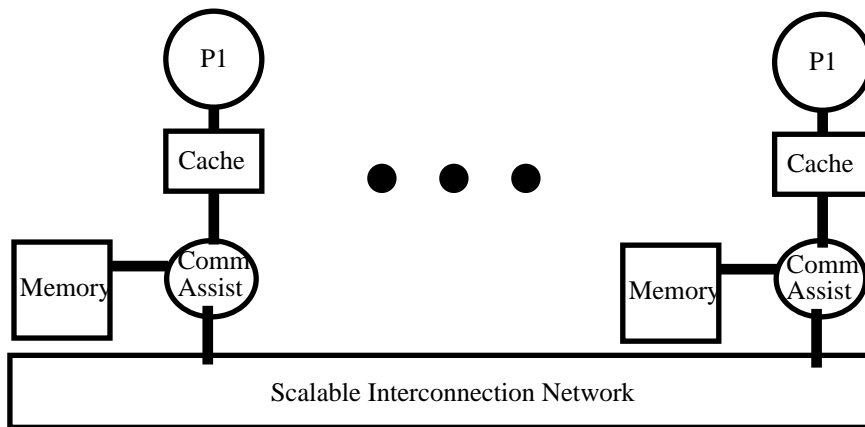
Figure 8-1 A generic scalable multiprocessor.

chapter will focus on full hardware support for this programming model, so in terms of the layers of abstraction it is supported directly at the hardware-software interface. Other programming models can be implemented in software on the systems discussed in this chapter (see Figure 8-2).
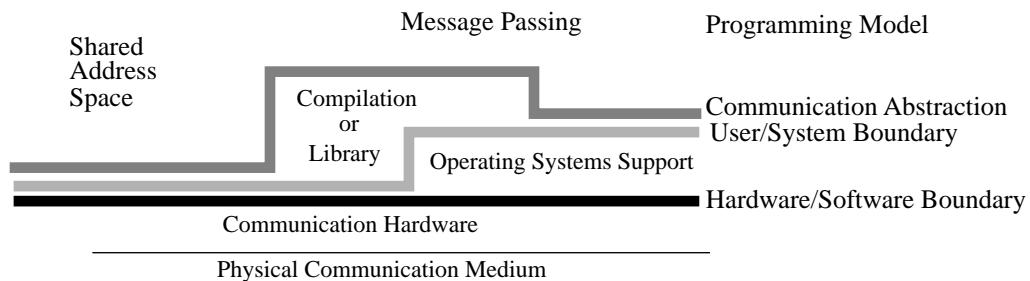
Figure 8-2 Layers of abstraction for systems discussed in this chapter.

The most popular approach to scalable cache coherence is based on the concept of a *directory*. Since the state of a block in the caches can no longer be determined implicitly, by placing a request on a shared bus and having it be snooped, the idea is to maintain this state explicitly in a place (a directory) where requests can go and look it up. Consider a simple example. Imagine that each block of main memory has associated with it, at the main memory, a record of the caches containing a copy of the block and the state therein. This record is called the directory entry for that block (see Figure 8-3). As in bus-based systems, there may be many caches with a clean,
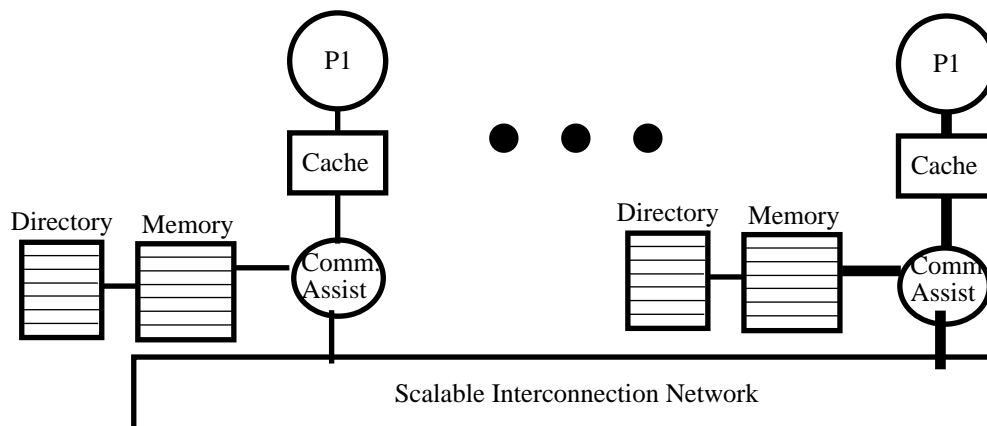
Figure 8-3 A scalable multiprocessor with directories.

readable block, but if the block is writable, possibly dirty, in one cache then only that cache may have a valid copy. When a node incurs a cache miss, it first communicates with the directory entry for the block using network transactions. Since the directory entry is collocated with the main memory for the block, its location can be determined from the address of the block. From the directory, the node determines where the valid copies (if any) are and what further actions to take. It then communicates with the copies as necessary using additional network transactions. The resulting changes to the state of cached blocks are also communicated to the directory entry for the block through network transactions, so the directory stays up to date.

On a read miss, the directory indicates from which node the data may be obtained, as shown in Figure 8-4(a). On a write miss, the directory identifies the copies of the block, and invalidation or update network transactions may be sent to these copies (Figure 8-4(b)). (Recall that a write to a block in shared state is also considered a write miss). Since invalidations or updates are sent to multiple copies through potentially disjoint paths in the network, determining the completion of a write now requires that all copies reply to invalidations with explicit acknowledgment transactions; we cannot assume completion when the read-exclusive or update request obtains access to the interconnect as we did on a shared bus. Requests, replies, invalidations, updates and acknowledgments across nodes are all network transactions like those of the previous chapter; only, here the end-point processing at the destination of the transaction (invalidating blocks, retrieving and replying with data) is typically done by the communication assist rather than the main processor. Since directory schemes rely on point to point network transactions, they can be used with any interconnection network. Important questions for directories include the actual form in which the directory information is stored, and how correct, efficient protocols may be designed around these representations.

A different approach than directories is to extend the broadcast and snooping mechanism, using a hierarchy of broadcast media like buses or rings. This is conceptually attractive because it builds larger systems hierarchically out of existing small-scale mechanisms. However, it does not apply to general networks such as meshes and cubes, and we will see that it has problems with latency and bandwidth, so it has not become very popular. One approach that is popular is a two-level protocol. Each node of the machine is itself a multiprocessor, and the caches within the node are kept coherent by one coherence protocol called the *inner* protocol. Coherence across nodes is maintained by another, potentially different protocol called the *outer* protocol. To the outer proto-

(a) Read miss to a block in dirty state

(b) Write miss to a block with two sharers
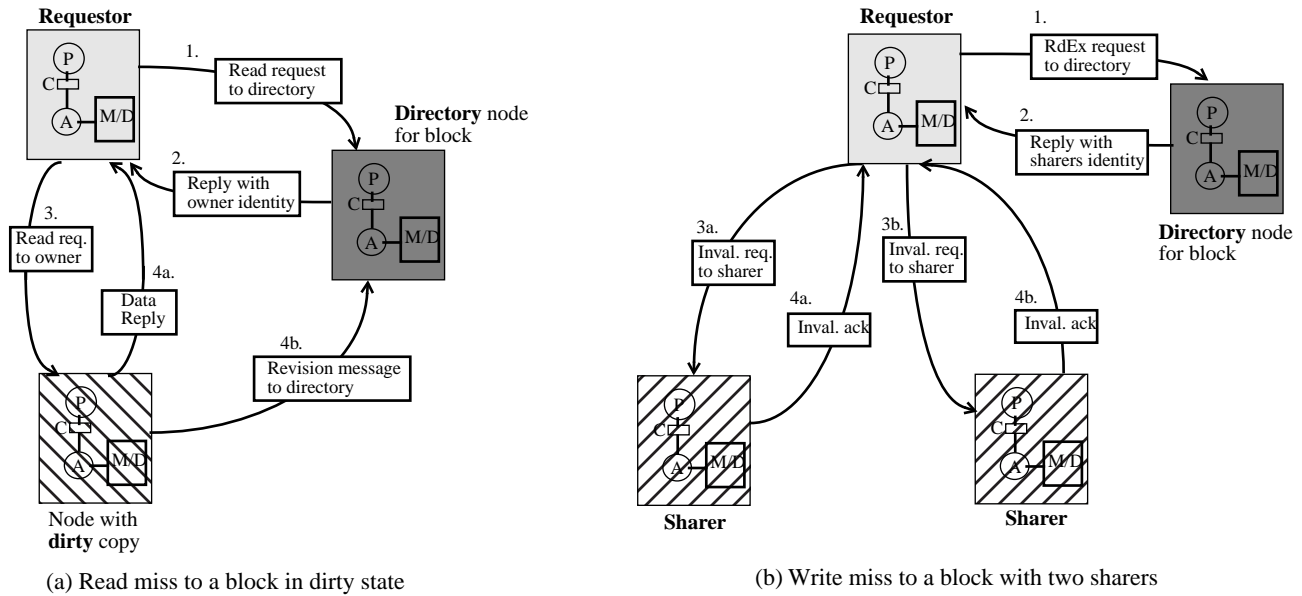
Figure 8-4 Basic operation of a simple directory.

Two example operations are shown. On the left is a read miss to a block that is currently held in dirty state by a node that is not the requestor or the node that holds the directory information. A read miss to a block that is clean in main memory (i.e. at the directory node) is simpler,: the main memory simply replies to the requestor and the miss is satisfied in a single request-reply pair. of transactions. On the right is a write miss to a block that is currently in shared state in two other nodes' caches (the two sharers). The big rectangles are the are nodes, and the arcs (with boxed labels) are network transactions. The numbers 1, 2 etc. next to a transaction show the serialization of transactions. Different letters next to the same number indicate that the transactions can be overlapped.

col each multiprocessor node looks like a single cache, and coherence within the node is the responsibility of the inner protocol. Usually, an adapter or a shared tertiary cache is used to represent a node to the outer protocol. The most common organization is for the outer protocol to be a directory protocol and the inner one to be a snoopy protocol [LoC96, LLJ+93, ClA96, WGH+97]. However, other combinations such as snoopy-snoopy [FBR93], directory-directory [Con93], and even snoopy-directory may be used.

Putting together smaller-scale machines to build larger machines in a two-level hierarchy is an attractive engineering option: It amortizes fixed per-node costs, may take advantage of packaging hierarchies, and may satisfy much of the interprocessor communication within a node and have only a small amount require more costly communication across nodes. The major focus of this chapter will be on directory protocols across nodes, regardless of whether the node is a uniprocessor or a multiprocessor or what coherence method it uses. However, the interactions among two-level protocols will be discussed. As we examine the organizational structure of the directory, the protocols involved in supporting coherence and consistency, and the requirements placed on the communication assist, we will find another rich and interesting design space.

The next section provides a framework for understanding different approaches for providing coherent replication in a shared address space, including snooping, directories and hierarchical snooping. Section 8.3 introduces a simple directory representation and a protocol corresponding to it, and then provides an overview of the alternatives for organizing directory information and

(a) Snooping-snooping

(b) Snooping-directory

(c) Directory-directory

(d) Directory-snooping
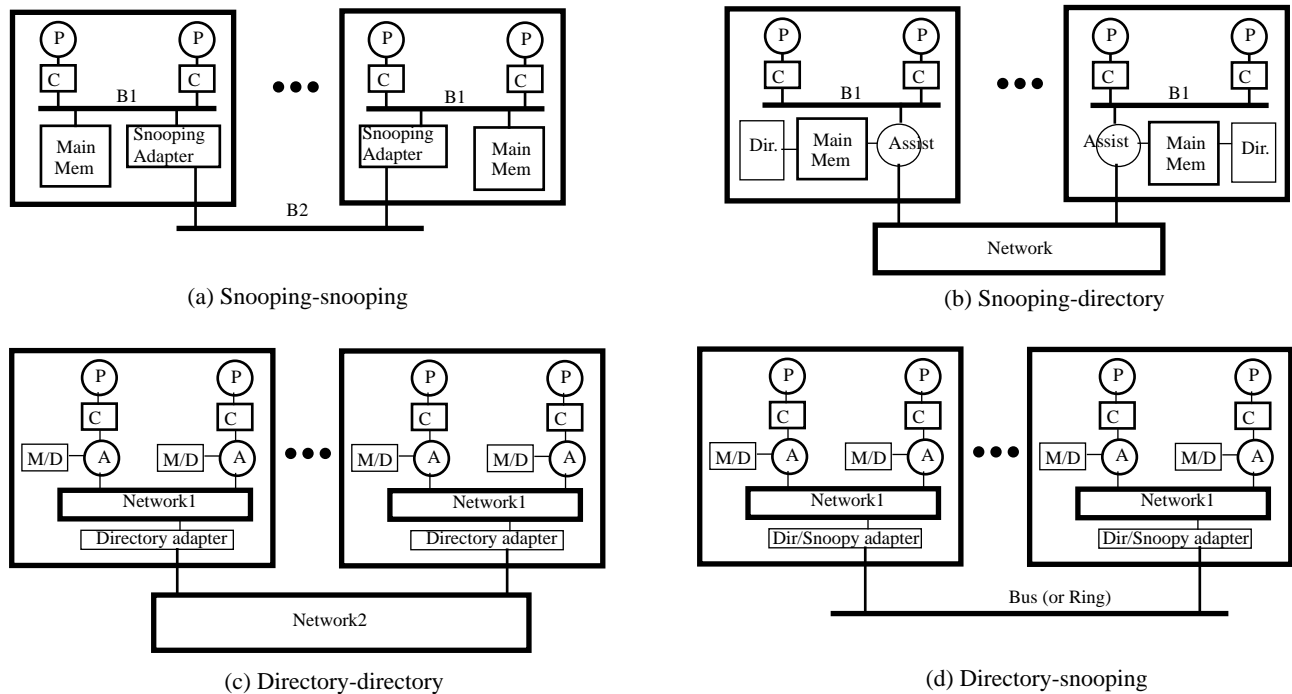
Figure 8-5 Some possible organizations for two-level cache-coherent systems.

Each node visible at the outer level is itself a multiprocessor. B1 is a first-level bus, and B2 is second-level bus. The label snooping-directory, for example, means that a snooping protocol is used to maintain coherence within a multiprocessor node, and a directory protocol is used to maintain coherence across nodes.

protocols. This is followed by a quantitative assessment of some issues and architectural tradeoffs for directory protocols in Section 8.4.

The chapter then covers the issues and techniques involved in actually designing correct, efficient protocols. Section 8.5 discusses the major new challenges for scalable, directory-based systems introduced by multiple copies of data without a serializing interconnect. The next two sections delve deeply into the two most popular types of directory-based protocols, discussing various design alternatives and using two commercial architectures as case studies: the Origin2000 from Silicon Graphics, Inc. and the NUMA-Q from Sequent Computer Corporation. Synchronization for directory-based multiprocessors is discussed in Section 8.9, and the implications for parallel software in Section 8.10. Section 8.11 covers some advanced topics, including the approaches of hierarchically extending the snooping and directory approaches for scalable coherence.

8.2 Scalable Cache Coherence

Before we discuss specific protocols, this section briefly lays out the major organizational alternatives for providing coherent replication in the extended memory hierarchy, and introduces the basic mechanisms that any approach to coherence must provide, whether it be based on snoop-

ing, hierarchical snooping or directories. Different approaches and protocols discussed in the rest of the chapter will refer to the mechanisms described here.

On a machine with physically distributed memory, nonlocal data may be replicated either only in the processors caches or in the local main memory as well. This chapter assumes that data are replicated only in the caches, and that they are kept coherent in hardware at the granularity of cache blocks just as in bus-based machines. Since main memory is physically distributed and has nonuniform access costs to a processor, architectures of this type are often called cache-coherent, nonuniform memory access architectures or CC-NUMA. Alternatives for replicating in main memory will be discussed in the next chapter, which will discuss other hardware-software tradeoffs in scalable shared address space systems as well.

Any approach to coherence, including snoopy coherence, must provide certain critical mechanisms. First, a block can be in each cache in one of a number of states, potentially in different states in different caches. The protocol must provide these *cache states*, the *state transition diagram* according to which the blocks in different caches independently change states, and the set of *actions* associated with the state transition diagram. Directory-based protocols also have a *directory state* for each block, which is the state of the block at the directory. The protocol may be invalidation-based, update-based, or hybrid, and the stable cache states themselves are very often the same (MESI, MSI) regardless of whether the system is based on snooping or directories. The tradeoffs in the choices of states are very similar to those discussed in Chapter 5, and will not be revisited in this chapter. Conceptually, the cache state of a memory block is a vector containing its state in every cache in the system (even when it is not present in a cache, it may be thought of as being present but in the invalid state, as discussed in Chapter 5). The same state transition diagram governs the copies in different caches, though the current state of the block at any given time may be different in different caches. The state changes for a block in different caches are coordinated through transactions on the interconnect, whether bus transactions or more general network transactions.

Given a protocol at cache state transition level, a coherent system must provide mechanisms for managing the protocol. First, a mechanism is needed to determine when (on which operations) to invoke the protocol. This is done in the same way on most systems: through an *access fault* (cache miss) detection mechanism. The protocol is invoked if the processor makes an access that its cache cannot satisfy by itself; for example, an access to a block that is not in the cache, or a write access to a block that is present but in shared state. However, even when they use the same set of cache states, transitions and access fault mechanisms, approaches to cache coherence differ substantially in the mechanisms they provide for the following three important functions that may need to be performed when an access fault occurs:

(a) Finding out enough information about the state of the location (cache block) in other caches to determine what action to take,

(b) Locating those other copies, if needed, for example to invalidate them, and

(c) Communicating with the other copies, e.g. obtaining data from or invalidating/updating them.

In snooping protocols, all of the above functions were performed by the broadcast and snooping mechanism. The processor puts a "search" request on the bus, containing the address of the block, and other cache controllers snoop and respond. A broadcast method can be used in distributed machines as well; the assist at the node incurring the miss can broadcast messages to all others nodes, and their assists can examine the incoming request and respond as appropriate.

However, broadcast does not scale, since it generates a large amount of traffic (p network transactions on every miss on a p-node machine). Scalable approaches include hierarchical snooping and directory-based approaches.

In a hierarchical snooping approach, the interconnection network is not a single broadcast bus (or ring) but a tree of buses. The processors are in the bus-based snoopy multiprocessors at the leaves of the tree. Parent buses are connected to children by interfaces that snoop the buses on both sides and propagate relevant transactions upward or downward in the hierarchy. Main memory may be centralized at the root or distributed among the leaves. In this case, all of the above functions are performed by the hierarchical extension of the broadcast and snooping mechanism: A processor puts a "search" request on its bus as before, and it is propagated up and down the hierarchy as necessary based on snoop results. Hierarchical snoopy systems are discussed further in Section 8.11.2.

In the simple directory approach described above, information about the state of blocks in other caches is found by looking up the directory through network transactions. The location of the copies is also found from the directory, and the copies are communicated with using point to point network transactions in an arbitrary interconnection network, without resorting to broadcast. Important questions for directory protocols are how the directory information is actually organized, and how protocols might be structured around these organizations using network transactions. The directory organization influences exactly how the protocol addresses the three issues above. An overview of directory organizations and protocols is provided in the next section.

8.3 Overview of Directory-Based Approaches

This section begins by describing a simple directory scheme and how it might operate using cache states, directory states, and network transactions. It then discusses the issues in scaling directories to large numbers of nodes, provides a classification of scalable directory organizations, and discusses the basics of protocols associated with these organizations.

The following definitions will be useful throughout our discussion of directory protocols. For a given cache or memory block:

The *home* node is the node in whose main memory the block is allocated.

The *dirty* node is the node that has a copy of the block in its cache in modified (dirty) state. Note that the home node and the dirty node for a block may be the same.

The *exclusive* node is the node that has a copy of the block in its cache in an exclusive state, either dirty or exclusive-clean as the case may be (recall from Chapter 5 that clean-exclusive means this is the only valid cached copy and that the block in main memory is up to date). Thus, the dirty node is also the exclusive node.

The *local* node, or *requesting* node, is the node containing the processor that issues a request for the block.

The *owner* node is the node that currently holds the valid copy of a block and must supply the data when needed; this is either the home node, when the block is not in dirty state in a cache, or the dirty node.

Blocks whose home is local to the issuing processor are called *locally allocated* (or sometimes simply *local*) blocks, while all others are called *remotely allocated* (or *remote*).

Let us begin with the basic operation of directory-based protocols using a very simple directory organization.

8.3.1 Operation of a Simple Directory Scheme

As discussed earlier, a natural way to organize a directory is to maintain the directory information for a block together with the block in main memory, i.e. at the home node for the block. A simple organization for the directory information for a block is as a bit vector. The bit vector contains *p presence bits* and one or more state bits for a *p*-node machine (see Figure 8-6). Each pres-
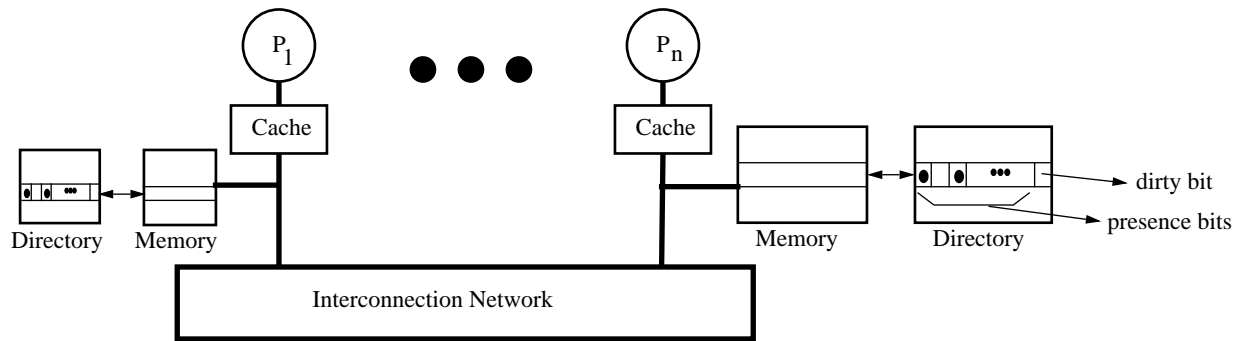
Figure 8-6 Directory information for a distributed-memory multiprocessor.

ence bit represents a particular node (uniprocessor or multiprocessor), and specifies whether or not that node has a cached copy of the block. Let us assume for simplicity that there is only one state bit, called the *dirty* bit, which indicates if the block is dirty in one of the node caches. Of course, if the dirty bit is ON, then only one node (the dirty node) should be caching that block and only that node's presence bit should be ON. With this structure, a read miss can easily determine from looking up the directory which node, if any, has a dirty copy of the block or if the block is valid in main memory, and a write miss can determine which nodes are the sharers that must be invalidated.

In addition to this directory state, state is also associated with blocks in the caches themselves, as discussed in Section 8.2, just as it was in snoopy protocols. In fact, the directory information for a block is simply main memory's view of the state of that block in different caches. The directory does not necessarily need to know the exact state (e.g. MESI) in each cache, but only enough information to determine what actions to take. The number of states at the directory is therefore typically smaller than the number of cache states. In fact, since the directory and the caches communicate through a distributed interconnect, there will be periods of time when a directory's knowledge of a cache state is incorrect since the cache state has been modified but notice of the modification has not reached the directory. During this time the directory may send a message to the cache based on its old (no longer valid) knowledge. The race conditions caused by this distri-

bution of state makes directory protocols interesting, and we will see how they are handled using transient states or other means in Sections 8.5 through 8.7.

To see how a read miss and write miss might interact with this bit vector directory organization, consider a protocol with three stable cache states (MSI) for simplicity, a single level of cache per processor, and a single processor per node. On a read or a write miss at node i (including an upgrade from shared state), the communication assist or controller at the local node i looks up the address of the memory block to determine if the home is local or remote. If it is remote, a network transaction is generated to the home node for the block. There, the directory entry for the block is looked up, and the controller at the home may treat the miss as follows, using network transactions similar to those that were shown in Figure 8-4:

- If the dirty bit is OFF, then the controller obtains the block from main memory, supplies it to the requestor in a reply network transaction, and turns the ith presence bit, *presence[i]*, ON.
- If the dirty bit is ON, then the controller replies to the requestor with the identity of the node whose presence bit is ON, i.e. the owner or dirty node. The requestor then sends a request network transaction to that owner node. At the owner, the cache changes its state to shared, and supplies the block to both the requesting node, which stores the block in its cache in shared state, as well as to main memory at the home node. At memory, the dirty bit is turned OFF, and *presence[i]* is turned ON.

A write miss by processor i goes to memory and is handled as follows:

- If the dirty bit is OFF, then main memory has a clean copy of the data. Invalidation request transactions are sent to all nodes j for which *presence[j]* is ON. The directory controller waits for invalidation acknowledgment transactions from these processors—indicating that the write has completed with respect to them—and then supplies the block to node i where it is placed in the cache in dirty state. The directory entry is cleared, leaving only *presence[i]* and the dirty bit ON. If the request is an upgrade instead of a read-exclusive, an acknowledgment is returned to the requestor instead of the data itself.
- If the dirty bit is ON, then the block is first recalled from the dirty node (whose presence bit is ON), using network transactions. That cache changes its state to invalid, and then the block is supplied to the requesting processor which places the block in its cache in dirty state. The directory entry is cleared, leaving only *presence[i]* and the dirty bit ON.

On a writeback of a dirty block by node i, the dirty data being replaced are written back to main memory, and the directory is updated to turn OFF the dirty bit and *presence[i]*. As in bus-based machines, writebacks cause interesting race conditions which will be discussed later in the context of real protocols. Finally, if a block in shared state is replaced from a cache, a message may be sent to the directory to turn off the corresponding presence bit so an invalidation is not sent to this node the next time the block is written. This message is called a *replacement hint*; whether it is sent or not does not affect the correctness of the protocol or the execution.

A directory scheme similar to this one was introduced as early as 1978 [CeF78] for use in systems with a few processors and a centralized main memory. It was used in the S-1 multiprocessor project at Lawrence Livermore National Laboratories [WiC80]. However, directory schemes in one form or another were in use even before this. The earliest scheme was used in IBM mainframes, which had a few processors connected to a centralized memory through a high-bandwidth switch rather than a bus. With no broadcast medium to snoop on, a duplicate copy of the cache tags for each processor was maintained at the main memory, and it served as the directory.

Requests coming to the memory looked up all the tags to determine the states of the block in the different caches [Tan76, Tuc86].

The value of directories is that they keep track of which nodes have copies of a block, eliminating the need for broadcast. This is clearly very valuable on read misses, since a request for a block will either be satisfied at the main memory or the directory will tell it exactly where to go to retrieve the exclusive copy. On write misses, the value of directories over the simpler broadcast approach is greatest if the number of sharers of the block (to which invalidations or updates must be sent) is usually small and does not scale up quickly with the number of processing nodes. We might already expect this to be true from our understanding of parallel applications. For example, in a near-neighbor grid computation usually two and at most four processes should share a block at a partition boundary, regardless of the grid size or the number of processors. Even when a location is actively read and written by all processes in the application, the number of sharers to be invalidated at a write depends upon the temporal interleaving of reads and writes by processors. A common example is *migratory* data, which are read and written by one processor, then read and written by another processor, and so on (for example, a global sum into which processes accumulate their values). Although all processors read and write the location, on a write only one other processor—the previous writer—has a valid copy since all others were invalidated at the previous write.

Empirical measurements of program behavior show that the number of valid copies on most writes to shared data is indeed very small the vast majority of the time, that this number does not grow quickly with the number of processes in the application, and that the frequency of writes to widely shared data is very low. Such data for our parallel applications will be presented and analyzed in light of application characteristics in Section 8.4.1. As we will see, these facts also help us understand how to organize directories cost-effectively. Finally, even if all processors running the application have to be invalidated on most writes, directories are still valuable for writes if the application does not run on all nodes of the multiprocessor.

8.3.2 Scaling

The main goal of using directory protocols is to allow cache coherence to scale beyond the number of processors that may be sustained by a bus. Scalability issues for directory protocols include both performance and the storage overhead of directory information. Given a system with distributed memory and interconnect, the major scaling issues for performance are how the latency and bandwidth demands presented to the system by the protocol scale with the number of processors used. The bandwidth demands are governed by the number of network transactions generated per miss (times the frequency of misses), and latency by the number of these transactions that are in the critical path of the miss. These performance issues are affected both by the directory organization itself and by how well the flow of network transactions is optimized in the protocol itself (given a directory organization), as we shall see. Storage, however, is affected only by how the directory information is organized. For the simple bit vector organization, the number of presence bits needed scales linearly with both the number of processing nodes (p bits per memory block) and the amount of main memory (one bit vector per memory block), leading to a potentially large storage overhead for the directory. With a 64-byte block size and 64 processors, the directory storage overhead is 64 bits divided by 64 bytes or 12.5 percent, which is not so bad. With 256 processors and the same block size, the overhead is 50 percent, and with 1024 processors it is 200 percent! The directory overhead does not scale well, but it may be acceptable if the number of nodes visible to the directory at the target machine scale is not very large.

Fortunately, there are many other ways to organize directory information that improve the scalability of directory storage. The different organizations naturally lead to different high-level protocols with different ways of addressing the protocol issues (a) through (c) and different performance characteristics. The rest of this section lays out the space of directory organizations and briefly describes how individual read and write misses might be handled in straightforward request-reply protocols that use them. As before, the discussion will pretend that there are no other cache misses in progress at the time, hence no race conditions, so the directory and the caches are always encountered as being in stable states. Deeper protocol issues—such as performance optimizations in structuring protocols, handling race conditions caused by the interactions of multiple concurrent memory operations, and dealing with correctness issues such as ordering, deadlock, livelock and starvation—will be discussed later when we plunge into the depths of directory protocols in Sections 8.5 through 8.7.

8.3.3 Alternatives for Organizing Directories

At the highest level, directory schemes are differentiated by the mechanisms they provide for the three functions of coherence protocols discussed in Section 8.2. Since communication with cached copies is always done through network transactions, the real differentiation is in the first two functions: finding the source of the directory information upon a miss, and determining the locations of the relevant copies.

There are two major classes of alternatives for finding the source of the directory information for a block:

- *Flat directory schemes*, and
- *Hierarchical directory schemes*.

The simple directory scheme described above is a flat scheme. Flat schemes are so named because the source of the directory information for a block is in a fixed place, usually at the home that is determined from the address of the block, and on a miss a request network transaction is sent directly to the home node to look up the directory in a single direct network transaction. In hierarchical schemes, memory is again distributed with the processors, but the directory information for each block is logically organized as a hierarchical data structure (a tree). The processing nodes, each with its portion of memory, are at the leaves of the tree. The internal nodes of the tree are simply hierarchically maintained directory information for the block: A node keeps track of whether each of its children has a copy of a block. Upon a miss, the directory information for the block is found by traversing up the hierarchy level by level through network transactions until a directory node is reached that indicates its subtree has the block in the appropriate state. Thus, a processor that misses simply sends a "search" message up to its parent, and so on, rather than directly to the home node for the block with a single network transaction. The directory tree for a block is logical, not necessarily physical, and can be embedded in any general interconnection network. In fact, in practice every processing node in the system is not only a leaf node for the blocks it contains but also stores directory information as an internal tree node for other blocks.

In the hierarchical case, the information about locations of copies is also maintained through the hierarchy itself; copies are found and communicated with by traversing up and down the hierarchy guided by directory information. In flat schemes, how the information is stored varies a lot. At the highest level, flat schemes can be divided into two classes:

- *memory-based* schemes, in which all the directory information about all cached copies is stored at the home node of the block. The basic bit vector scheme described above is memory-based. The locations of all copies are discovered at the home, and they can be communicated with directly through point-to-point messages.

- *cache-based* schemes, in which the information about cached copies is not all contained at the home but is distributed among the copies themselves. The home simply contains a pointer copy of the block. Each cached copy then contains a pointer to (or the identity of) the node that has the next cached copy of the block, in a distributed linked list organization. The locations of copies are therefore determined by traversing this list via network transactions.

Figure 8-7 summarizes the taxonomy. Hierarchical directories have some potential advantages. For example, a read miss to a block whose home is far away in the interconnection network topology might be satisfied closer to the issuing processor if another copy is found nearby in the topology as the request traverses up and down the hierarchy, instead of having to go all the way to the home. Also, requests from different nodes can potentially be combined at a common ancestor in the hierarchy, and only one request sent on from there. These advantages depend on how well the logical hierarchy matches the underlying physical network topology. However, instead of only a few point-to-point network transactions needed to satisfy a miss in many flat schemes, the *number* of network transactions needed to traverse up and down the hierarchy can be much larger, and each transaction along the way needs to look up the directory information at its destination node. As a result, the latency and bandwidth requirements of hierarchical directory schemes tend to be much larger, and these organizations are not popular on modern systems. Hierarchical directories will not be discussed much in this chapter, but will be described briefly together with hierarchical snooping approaches in Section 8.11.2. The rest of this section will examine flat directory schemes, both memory-based and cache-based, looking at directory organizations, storage overhead, the structure of protocols, and the impact on performance characteristics.

Directory Storage Schemes

Finding source of directory information

Flat   **Centralized**   **Hierarchical**
(Hierarchy of caches that guarantee the inclusion property. Each parent keeps track of exactly which of its immediate children have a copy of the block.)
Examples: Data Diffusion Machine.

Locating Copies

Memory-based   **Cache-based**
(Directory information collocated with memory module that is home for that memory block)   (Caches holding a copy of the memory block form a linked list; memory holds pointer to head of linked list)

Examples: Stanford DASH/FLASH, MIT Alewife, SGI Origin, HAL.   Examples: IEEE SCI, Sequent NUMA-Q.

Figure 8-7 Alternatives for storing directory information.

Flat, Memory-Based Directory Schemes

The bit vector organization above, called a *full bit vector* organization, is the most straightforward way to store directory information in a flat, memory based scheme The style of protocol that results has already been discussed. Consider its basic performance characteristics on writes. Since it preserves information about sharers precisely and at the home, the number of network transactions per invalidating write grows only with the number of sharers. Since the identity of all sharers is available at the home, invalidations sent to them can be overlapped or even sent in parallel; the number of fully serialized network transactions in the critical path is thus not proportional to the number of sharers, reducing latency. The main disadvantage, as discussed earlier, is storage overhead.

There are two ways to reduce the overhead of full bit vector schemes. The first is to increase the cache block size. The second is to put multiple processors, rather than just one, in a node that is visible to the directory protocol; that is, to use a two-level protocol. The Stanford DASH machine uses a full bit vector scheme, and its nodes are 4-processor bus-based multiprocessors. These two methods actually make full-bit-vector directories quite attractive for even fairly large machines. For example, for a 256-processor machine using 4-processor nodes and 128-byte cache blocks, the directory memory overhead is only 6.25%. As small-scale SMPs become increasingly attractive building blocks, this storage problem may not be severe.

However, these methods reduce the overhead by only a small constant factor each. The total storage overhead is still proportional to $P*M$, where P is the number of processing nodes and M is the number of total memory blocks in the machine ($M=P*m$, where m is the number of blocks per local memory), and would become intolerable in very large machines. Directory storage overhead can be reduced further by addressing each of the orthogonal factors in the $P*M$ expression. We can reduce the *width* of each directory entry by not letting it grow proportionally to P. Or we can reduce the total number of directory entries, or directory *height*, by not having an entry per memory block.

Directory width is reduced by using *limited pointer* directories, which are motivated by the earlier observation that most of the time only a few caches have a copy of a block when the block is written. Limited pointer schemes therefore do not store yes or no information for all nodes, but rather simply maintain a fixed number of pointers, say i, each pointing to a node that currently caches a copy of the block [ASH+88]. Each pointer takes $\log P$ bits of storage for P nodes, but the number of pointers used is small. For example, for a machine with 1024 nodes, each pointer needs 10 bits, so even having 100 pointers uses less storage than a full bit-vector scheme. In practice, five or less pointers seem to suffice. Of course, these schemes need some kind of backup or overflow strategy for the situation when more than i copies are cached, since they can keep track of only i copies precisely. One strategy may be to resort to broadcasting invalidations to all nodes when there are more than i copies. Many other strategies have been developed to avoid broadcast even in these cases. Different limited pointer schemes differ primarily in their overflow strategies and in the number of pointers they use.

Directory height can be reduced by organizing the directory as a cache, taking advantage of the fact that since the total amount of cache in the machine is much smaller than the total amount of memory, only a very small fraction of the memory blocks will actually be replicated in caches at a given time, so most of the directory entries will be unused anyway [GWM90, OKN90]. The

Advanced Topics section of this chapter (Section 8.11) will discuss the techniques to reduce directory width and height in more detail.

Regardless of these optimizations, the basic approach to finding copies and communicating with them (protocol issues (b) and (c)) is the same for the different flat, memory-based schemes. The identities of the sharers are maintained at the home, and at least when there is no overflow the copies are communicated with by sending point to point transactions to each.

Flat, Cache-Based Directory Schemes

In flat, cache-based schemes there is still a home main memory for the block; however, the directory entry at the home does not contain the identities of all sharers, but only a pointer to the first sharer in the list plus a few state bits. This pointer is called the *head pointer* for the block. The remaining nodes caching that block are joined together in a *distributed, doubly linked list*, using additional pointers that are associated with each cache line in a node (see Figure 8-8). That is, a cache that contains a copy of the block also contains pointers to the next and previous caches that have a copy, called the *forward* and *backward* pointers, respectively.

On a read miss, the requesting node sends a network transaction to the home memory to find out the identity of the head node of the linked list, if any, for that block. If the head pointer is null (no current sharers) the home replies with the data. If the head pointer is not null, then the requestor must be added to the list of sharers. The home replies to the requestor with the head pointer. The requestor then sends a message to the head node, asking to be inserted at the head of the list and hence become the new head node. The net effect is that the head pointer at the home now points to the requestor, the forward pointer of the requestor's own cache entry points to the old head node (which is now the second node in the linked list), and the backward pointer of the old head node point to the requestor. The data for the block is provided by the home if it has the latest copy, or by the head node which always has the latest copy (is the owner) otherwise.

On a write miss, the writer again obtains the identity of the head node, if any, from the home. It then inserts itself into the list as the head node as before (if the writer was already in the list as a sharer and is now performing an upgrade, it is deleted from its current position in the list and inserted as the new head). Following this, the rest of the distributed linked list is traversed node-to-node via network transactions to find and invalidate successive copies of the block. If a block that is written is shared by three nodes A, B and C, the home only knows about A so the writer sends an invalidation message to it; the identity of the next sharer B can only be known once A is reached, and so on. Acknowledgments for these invalidations are sent to either the writer. Once again, if the data for the block is needed it is provided by either the home or the head node as appropriate. The number of messages per invalidating write—the bandwidth demand—is proportional to the number of sharers as in the memory-based schemes, but now so is the number of messages in the critical path, i.e. the latency. Each of these serialized messages also invokes the communication assist, increasing latency further. In fact, even a read miss to a clean block involves the controllers of multiple nodes to insert the node in the linked list.

Writebacks or other replacements from the cache also require that the node delete itself from the sharing list, which means communicating with the nodes before and after it in the list. This is necessary because the new block that replaces the old one will need the forward and backward pointer slots of the cache entry for its own sharing list. An example cache-based protocol will be described in more depth in Section 8.7.
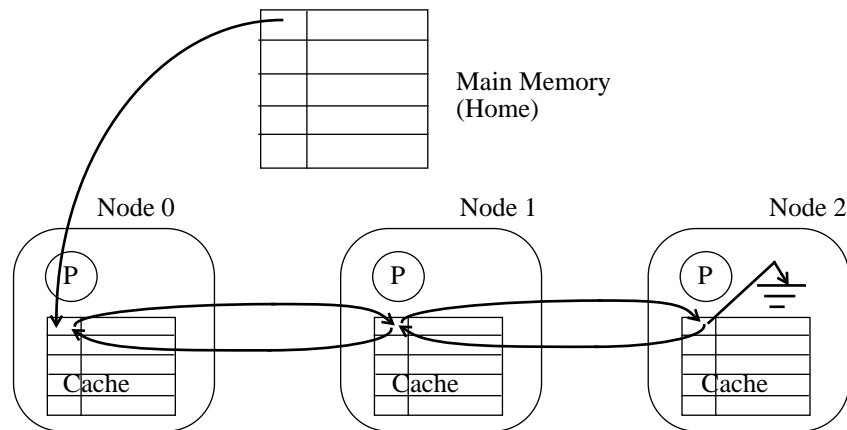
Figure 8-8 The doubly linked list distributed directory representation in the IEEE Scalable Coherent Interface (SCI) standard.

A cache line contains not only data and state for the block, but also forward and backward pointers for the distributed linked list.

To counter the latency disadvantage, cache-based schemes have some important advantages over memory-based schemes. First, the directory overhead is not so large to begin with. Every block in main memory has only a single head pointer. The number of forward and backward pointers is proportional to the number of cache blocks in the machine, which is much smaller than the number of memory blocks. The second advantage is that a linked list records the order in which accesses were made to memory for the block, thus making it easier to provide fairness and avoid live-lock in a protocol (most memory-based schemes do not keep track of request order in their sharing vectors or lists, as we will see). Third, the work to be done by controllers in sending invalidations is not centralized at the home but rather distributed among sharers, thus perhaps spreading out controller occupancy and reducing the corresponding bandwidth demands placed at the home.

Manipulating insertion in and deletion from distributed linked lists can lead to complex protocol implementations. For example, deleting a node from a sharing list requires careful coordination and mutual exclusion with processors ahead and behind it in the linked list, since those processors may also be trying to replacing the same block concurrently. These complexity issues have been greatly alleviated by the formalization and publication of a standard for a cache-based directory organization and protocol: the IEEE 1596-1992 *Scalable Coherence Interface* standard (SCI) [Gus92]. The standard includes a full specification and C code for the protocol. Several commercial machines have started using this protocol (e.g. Sequent NUMA-Q [LoC96], Convex Exemplar [Con93,TSS+96], and Data General [ClA96]), and variants that use alternative list representations (singly-linked lists instead of the doubly-linked lists in SCI) have also been explored [ThD90]. We shall examine the SCI protocol itself in more detail in Section 8.7, and defer detailed discussion of advantages and disadvantages until then.

Summary of Directory Organization Alternatives

To summarize, there are clearly many different ways to organize how directories store the cache state of memory blocks. Simple bit vector representations work well for machines that have a moderate number of nodes visible to the directory protocol. For larger machines, there are many alternatives available to reduce the memory overhead. The organization chosen does, however, affect the complexity of the coherence protocol and the performance of the directory scheme against various sharing patterns. Hierarchical directories have not been popular on real machines, while machines with flat memory-based and cache-based (linked list) directories have been built and used for some years now.

This section has described the basic organizational alternatives for directory protocols and how the protocols work a high level. The next section quantitatively assesses the appropriateness of directory-based protocols for scalable multiprocessors, as well as some important protocol and architectural tradeoffs at this basic level. After this, the chapter will dive deeply into examining how memory-based and cache-based directory protocols might actually be realized, and how they might address the various performance and correctness issues that arise.

8.4 Assessing Directory Protocols and Tradeoffs

Like in Chapter 5, this section uses a simulator of a directory-based protocol to examine some relevant characteristics of applications that can inform architectural tradeoffs, but that cannot be measured on these real machines. As mentioned earlier, issues such as three-state versus four-state or invalidate versus update protocols that were discussed in Chapter 5 will not be revisited here. (For update protocols, an additional disadvantage in scalable machines is that useless updates incur a separate network transaction for each destination, rather than a single bus transaction that is snooped by all caches. Also, we will see later in the chapter that besides the performance tradeoffs, update-based protocols make it much more difficult to preserve the desired memory consistency model in directory-based systems.) In particular, this section quantifies the distribution of invalidation patterns in directory protocols that we referred to earlier, examines how the distribution of traffic between local and remote changes as the number of processors is increased for a fixed problem size, and revisits the impact of cache block size on traffic which is now divided into local, remote and overhead. In all cases, the experiments assume a memory-based flat directory protocol with MESI cache states, much like that of the Origin2000.

8.4.1 Data Sharing Patterns for Directory Schemes

It was claimed earlier that the number of invalidations that need to be sent out on a write is usually small, which makes directories particularly valuable. This section quantifies that claim for our parallel application case studies, develops a framework for categorizing data structures in terms of sharing patterns and understanding how they scale, and explains the behavior of the application case studies in light of this framework.

Sharing Patterns for Application Case Studies

In an invalidation-based protocol, two aspects of an application's data sharing patterns are important to understand in the context of directory schemes: (i) the frequency with which processors

issue writes that may require invalidating other copies (i.e. writes to data that are not dirty in the writer's cache, or *invalidating writes*), called the *invalidation frequency*, and (ii) the distribution of the number of invalidations (sharers) needed upon these writes, called the *invalidation size distribution*. Directory schemes are particularly advantageous if the invalidation size distribution has small measures of average, and as long as the frequency is significant enough that using broadcast all the time would indeed be a performance problem. Figure 8-9 shows the invalidation
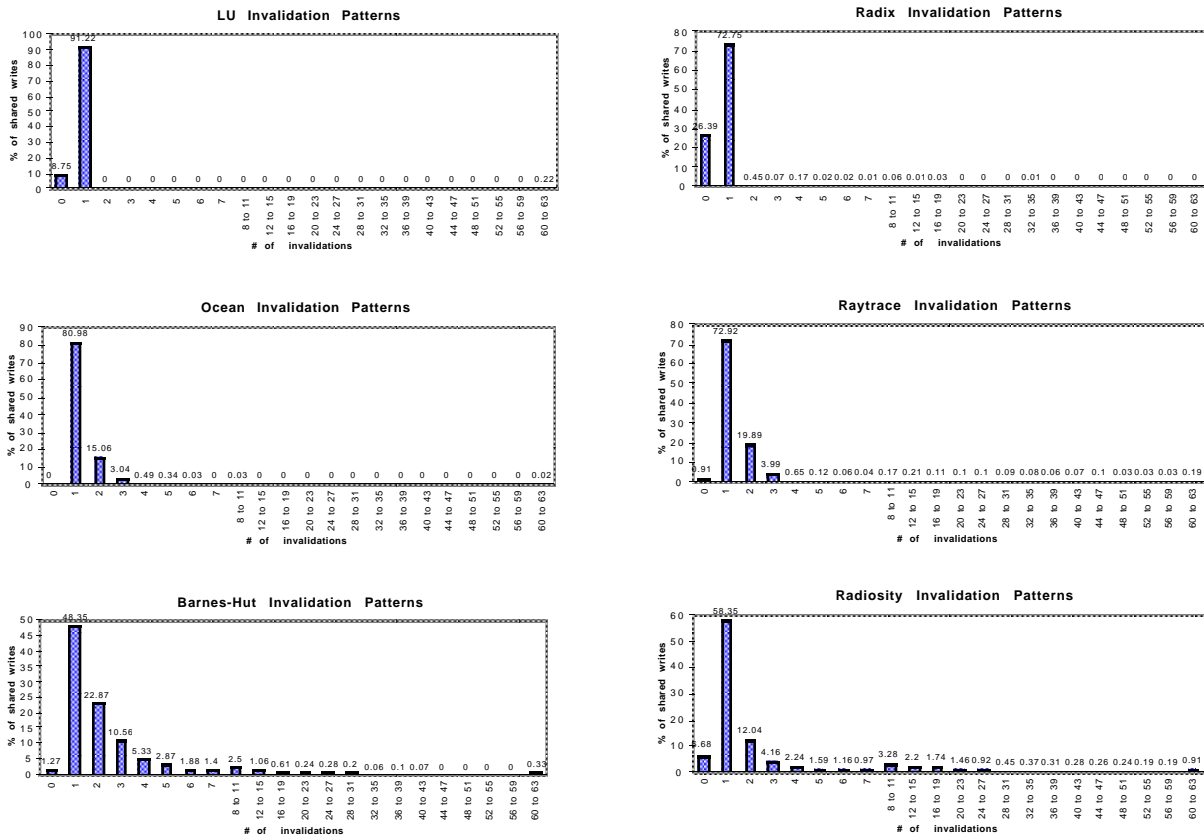
Figure 8-9 Invalidation patterns with default data sets and 64 processors.

The x-axis shows the invalidation size (number of active sharers) upon an invalidating write, and the y-axis shows the frequency of invalidating writes for each x-axis value. The invalidation size distribution was measured by simulating a full bit vector directory representation and recording for every invalidating write how many presence bits (other than the writer's) are set when the write occurs. Frequency is measured as invalidating writes with that many invalidations per 1000 instructions issued by the processor. The data are averaged over all the 64 processors used.

size distributions for our parallel application case studies running on 64-node systems for the default problem sizes presented in Chapter 4. Infinite per-processor caches are used in these simulations, to capture inherent sharing patterns. With finite caches, replacement hints sent to the directory may turn off presence bits and reduce the number of invalidations sent on writes in some cases (though not the traffic since the replacement hint has to be sent).

It is clear that the measures of average of the invalidation size distribution are small, indicating that directories are likely to be very useful in containing traffic and that it is not necessary for the directory to maintain a presence bit per processor in a flat memory-based scheme. The non-zero

frequencies at very large processor counts are usually due to synchronization variables, where many processors spin on a variable and one processor writes it invalidating them all. We are interested in not just the results for a given problem size and number of processors, but also in how they scale. The communication to computation ratios discussed in Chapter 4 give us a good idea about how the frequency of invalidating writes should scale. For the size distributions, we can appeal to our understanding of the applications, which can also help explain the basic results observed in Figure 8-9 as we will see shortly.

A Framework for Sharing Patterns

More generally, we can group sharing patterns for data structures into commonly encountered categories, and use this categorization to reason about the resulting invalidation patterns and how they might scale. Data access patterns can be categorized in many ways: predictable versus unpredictable, regular versus irregular, coarse-grained versus fine-grained (or contiguous versus non-contiguous in the address space), near-neighbor versus long-range, etc. For the current purpose, the relevant categories are read-only, producer-consumer, migratory, and irregular read-write.

> *Read-only* Read-only data structures are never written once they have been initialized. There are no invalidating writes, so these data are a non-issue for directories. Examples include program code and the scene data in the Raytrace application.

> *Producer-consumer* A processor produces (writes) a data item, then one or more processors consume (read) it, then a processor produces it again, and so on. Flag-based synchronization is an example, as is the near-neighbor sharing in an iterative grid computation. The producer may be the same process every time, or it may change; for example in a branch-and-bound algorithm the bound may be written by different processes as they find improved bounds. The invalidation size is determined by how many consumers there have been each time the producer writes the value. We can have situations with one consumer, all processes being consumers, or a few processes being consumers. These situations may have different frequencies and scaling properties, although for most applications studied so far either the size does not scale quickly with the number of processors or the frequency has been found to be low.[1]

> *Migratory* Migratory data bounce around or migrate from one processor to another, being written (and usually read) by each processor to which they bounce. An example is a global sum, into which different processes add their partial sums. Each time a processor writes the variable only the previous writer has a copy (since it invalidated the previous "owner" when it did its write), so only a single invalidation is generated upon a write regardless of the number of processors used.

> *Irregular Read-Write* This corresponds to irregular or unpredictable read and write access patterns to data by different processes. A simple example is a distributed task queue system.

---

1. Examples of the first situation are the non-corner border elements in a near-neighbor regular grid partition and the key permutations in Radix. They lead to an invalidation size of one, which does not increase with the number of processors or the problem size. Examples all processes being consumers are a global energy variable that is read by all processes during a time-step of a physical simulation and then written by one at the end, or a synchronization variable on which all processes spin. While the invalidation size here is large, equal to the number of processors, such writes fortunately tend to happen very infrequently in real applications. Finally, examples of a few processes being consumers are the corner elements of a grid partition, or the flags used for tree-based synchronization. This leads to an invalidation size of a few, which may or may not scale with the number of processors (it doesn't in these two examples).

Processes will probe (read) the head pointer of a task queue when they are looking for work to steal, and this head pointer will be written when a task is added at the head. These and other irregular patterns usually tend to lead to wide-ranging invalidation size distribution, but in most observed applications the frequency concentration tends to be very much toward the small end of the spectrum (see the Radiosity example in Figure 8-9).

A similar categorization can be found in [GuW92].

Applying the Framework to the Application Case Studies

Let us now look briefly at each of the applications in Figure 8-9 to interpret the results in light of the sharing patterns as classified above, and to understand how the size distributions might scale.

In the LU factorization program, when a block is written it has previously been read only by the same processor that is doing the writing (the process to which it is assigned). This means that no other processor should have a cached copy, and zero invalidations should be sent. Once it is written, it is read by several other processes and no longer written further. The reason that we see one invalidation being sent in the figure is that the matrix is initialized by a single process, so particularly with infinite caches that process has a copy of the entire matrix in its cache and will be invalidated the first time another processor does a write to a block. An insignificant number of invalidating writes invalidates all processes, which is due to some global variables and not the main matrix data structure. As for scalability, increasing the problem size or the number of processors does not change the invalidation size distribution much, except for the global variables. Of course, they do change the frequencies.

In the Radix sorting kernel, invalidations are sent in two producer-consumer situations. In the permutation phase, the word or block written has been read since the last write only by the process to which that key is assigned, so at most a single invalidation is sent out. The same key position may be written by different processes in different outer loop iterations of the sort; however, in each iteration there is only one reader of a key so even this infrequent case generates only two invalidations (one to the reader and one to the previous writer). The other situation is the histogram accumulation, which is done in a tree-structured fashion and usually leads to a small number of invalidations at a time. These invalidations to multiple sharers are clearly very infrequent. Here too, increasing the problem size does not change the invalidation size in either phase, while increasing the number of processors increases the sizes but only in some infrequent parts of the histogram accumulation phase. The dominant pattern by far remains zero or one invalidations.

The nearest- neighbor, producer-consumer communication pattern on a regular grid in Ocean leads to most of the invalidations being to zero or one processes (at the borders of a partition). At partition corners, more frequently encountered in the multigrid equation solver, two or three sharers may need to be invalidated. This does not grow with problem size or number of processors. At the highest levels of the multigrid hierarchy, the border elements of a few processors' partitions might fall on the same cache block, causing four or five sharers to be invalidated. There are also some global accumulator variables, which display a migratory sharing pattern (one invalidation), and a couple of very infrequently used one-producer, all-consumer global variables.

The dominant, scene data in Raytrace are read-only. The major read-write data are the image and the task queues. Each word in the image is written only once by one processor per frame. This leads to either zero invalidations if the same processor writes a given image pixel in consecutive

frames (as is usually the case), or one invalidation if different processors do, as might be the case when tasks are stolen. The task queues themselves lead to the irregular read-write access patterns discussed earlier, with a wide-ranging distribution that is dominated in frequency by the low end (hence the very small non-zeros all along the x-axis in this case). Here too there are some infrequently written one-producer, all-consumer global variables.

In the Barnes-Hut application, the important data are the particle and cell positions, the pointers used to link up the tree, and some global variables used as energy values. The position data are of producer-consumer type. A given particle's position is usually read by one or a few processors during the force-calculation (tree-traversal) phase. The positions (centers of mass) of the cells are read by many processes, the number increasing toward the root which is read by all. These data thus cause a fairly wide range of invalidation sizes when they are written by their assigned processor after force-calculation.The root and upper-level cells responsible for invalidations being sent to all processors, but their frequency is quite small. The tree pointers are quite similar in their behavior. The first write to a pointer in the tree-building phase invalidates the caches of the processors that read it in the previous force-calculation phase; subsequent writes invalidate those processors that read that pointer during tree-building, which irregularly but mostly to a small number of processors. As the number of processors is increased, the invalidation size distribution tends to shift to the right as more processors tend to read a given item, but quite slowly, and the dominant invalidations are still to a small number of processors. The reverse effect (also slow) is observed when the number of particles is increased.

Finally, the Radiosity application has very irregular access patterns to many different types of data, including data that describe the scene (patches and elements) and the task queues. The resulting invalidation patterns, shown in Figure 8-9, show a wide distribution, but with the greatest frequency by far concentrated toward zero to two invalidations. Many of the accesses to the scene data also behave in a migratory way, and there are a few migratory counters and a couple of global variables that are one-producer, all-consumer.

The empirical data and categorization framework indicate that in most cases the invalidation size distribution is dominated by small numbers of invalidations. The common use of parallel machines as multiprogrammed compute servers for sequential or small-way parallel applications further limits the number of sharers. Sharing patterns that cause large numbers of invalidations are empirically found to be very infrequent at runtime. One exception is highly contented synchronization variables, which are usually handled specially by software or hardware as we shall see. In addition to confirming the value of directories, this also suggests ways to reduce directory storage, as we shall see next.

8.4.2 Local versus Remote Traffic

For a given number of processors and machine organization, the fraction of traffic that is local or remote depends on the problem size. However, it is instructive to examine how the traffic and its distribution changes with number of processors even when the problem size is held fixed (i.e. under PC scaling). Figure 8-10 shows the results for the default problem sizes, breaking down the remote traffic into various categories such as sharing (true or false), capacity, cold-start, write-back and overhead. Overhead includes the fixed header per cache block sent across the network, including the traffic associated with protocol transactions like invalidations and acknowledgments that do not carry any data. This component of traffic is different than on a bus-based machine: each individual point-to-point invalidation consumes traffic, and acknowledgments
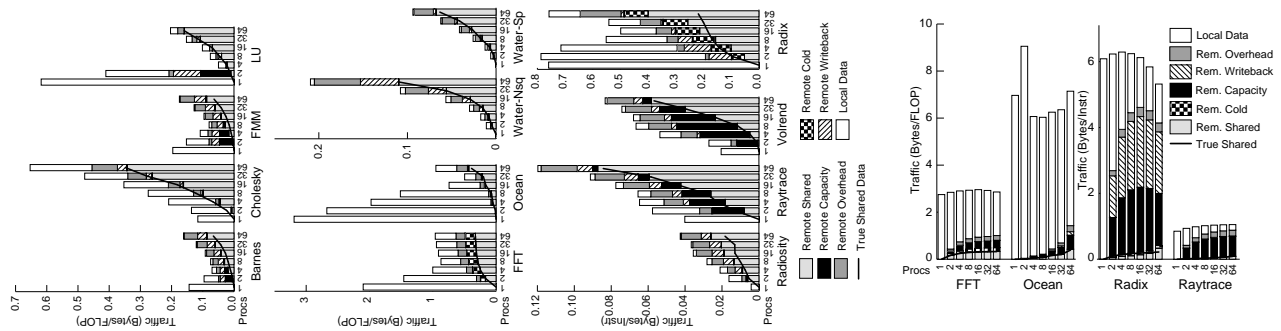
Figure 8-10 Traffic versus number of processors.

The overhead per block transferred across the network is 8 bytes. No overhead is considered for local accesses. The graphs on the left assume 1MB per-processor caches, and the ones on the right 64KB. The caches have a 64 byte block size and are 4-way set associative.

place traffic on the interconnect too. Here also, traffic is shown in bytes per FLOP or bytes per instruction for different applications. We can see that both local traffic and capacity-related remote traffic tend to decrease when the number of processors increases, due to both decrease in per-processor working sets and decrease in cold misses that are satisfied locally instead of remotely. However, sharing-related traffic increases as expected. In applications with small working sets, like Barnes-Hut, LU, and Radiosity, the fraction of capacity-related traffic is very small, at least beyond a couple of processors. In irregular applications like Barnes-Hut and Raytrace, most of the capacity-related traffic is remote, all the more so as the number of processors increases, since data cannot be distributed at page grain for it to be satisfied locally. However, in cases like Ocean, the capacity-related traffic is substantial even with the large cache, and is almost entirely local when pages are placed properly. With round-robin placement of shared pages, we would have seen most of the local capacity misses turn to remote ones.

When we use smaller caches to capture the realistic scenario of working sets not fitting in the cache in Ocean and Raytrace, we see that capacity traffic becomes much larger. In Ocean, most of this is still local, and the trend for remote traffic looks similar. Poor distribution of pages would have swamped the network with traffic, but with proper distribution network (remote) traffic is quite low. However, in Raytrace the capacity-related traffic is mostly remote, and the fact that it dominates changes the slope of the curve of total remote traffic: remote traffic still increases with the number of processors, but much more slowly since the working set size and hence capacity miss rate does not depend that much on the number of processors.

When a miss is satisfied remotely, whether it is satisfied at the home or needs another message to obtain it from a dirty node depends both on whether it is a sharing miss or a capacity/conflict/ cold miss, as well as on the size of the cache. In a small cache, dirty data may be replaced and written back, so a sharing miss by another processor may be satisfied at the home rather than at the (previously) dirty node. For applications that allow data to be placed in the memory of the node to which they are assigned, it is often the case that only that node writes the data, so even if the data are found dirty they are found so at the home itself. The extent to which this is true depends on both the application and whether the data are indeed distributed properly.

8.4.3 Cache Block Size Effects

The effects of block size on cache miss rates and bus traffic were assessed in Chapter 5, at least up to 16 processors. For miss rates, the trends beyond 16 processors extend quite naturally, except for threshold effects in the interaction of problem size, number of processors and block size as discussed in Chapter 4. This section examines the impact of block size on the components of local and remote traffic.
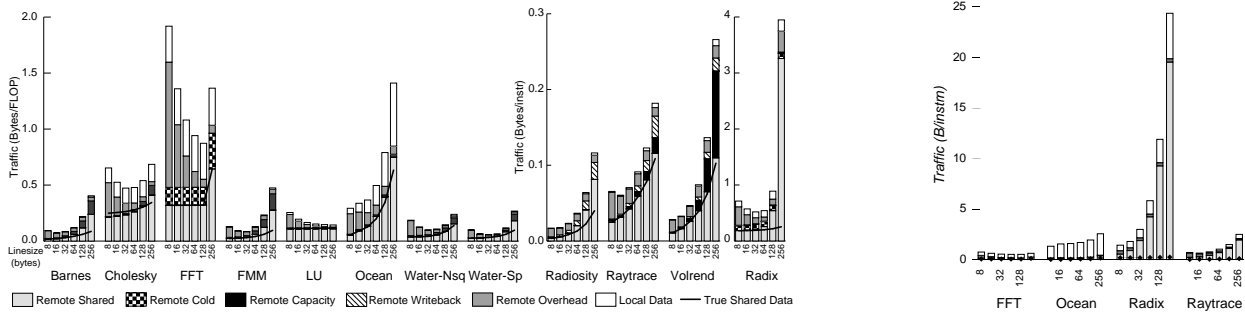
Figure 8-11 Traffic versus cache block size.

The overhead per block transferred over the network is 8 bytes. No overhead is considered for local access. The graph on the left shows the data for 1MB per processor caches, and that on the right for 64KB caches. All caches are 4-way set associative.

Figure 8-11 shows how traffic scales with block size for 32-processor executions of the applications with 1MB caches. Consider Barnes-Hut. The overall traffic increases slowly until about a 64 byte block size, and more rapidly thereafter primarily due to false sharing. However, the magnitude is small. Since the overhead per block moved through the network is fixed (as is the cost of invalidations and acknowledgments), the overhead component tends to shrink with increasing block size to the extent that there is spatial locality (i.e. larger blocks reduce the number of blocks transferred). LU has perfect spatial locality, so the data traffic remains fixed as block size increases. Overhead is reduced, so overall traffic in fact shrinks with increasing block size. In Raytrace, the remote capacity traffic has poor spatial locality, so it grows quickly with block size. In these programs, the true sharing traffic has poor spatial locality too, as is the case in Ocean at column-oriented partition borders (spatial locality even on remote data is good at row-oriented borders). Finally, the graph for Radix clearly shows the impact on remote traffic for false sharing when it occurs past the threshold block size (here about 128 or 256 bytes).

8.5 Design Challenges for Directory Protocols

Actually designing a correct, efficient directory protocol involves issues that are more complex and subtle than the simple organizational choices we have discussed so far, just as actually designing a bus-based protocol was more complex than choosing the number of states and drawing the state transition diagram for stable states. We had to deal with the non-atomicity of state transitions, split transaction busses, serialization and ordering issues, deadlock, livelock and starvation. Having understood the basics of directories, we are now ready to dive deeply into these issues for them as well. This section discusses the major new protocol-level design challenges that arise in implementing directory protocols correctly and with high performance, and identi-

fies general techniques for addressing these challenges. The next two sections will discuss case studies of memory-based and cache-based protocols in detail, describing the exact states and directory structures they use, and will examine the choices they make among the various techniques for addressing design challenges. They will also examine how some of the necessary functionality is implemented in hardware, and illustrate how the actual choices made are influenced by performance, correctness, simplicity, cost, and also limitations of the underlying processing node used.

As with the systems discussed in the previous two chapters, the design challenges for scalable coherence protocols are to provide high performance while preserving correctness, and to contain the complexity that results. Let us look at performance and correctness in turn, focusing on issues that were not already addressed for bus-based or non-caching systems. Since performance optimizations tend to increase concurrency and complicate correctness, let us examine them first.

8.5.1 Performance

The network transactions on which cache coherence protocols are built differ from those used in explicit message passing in two ways. First, they are automatically generated by the system, in particular by the communication assists or controllers in accordance with the protocol. Second, they are individually small, each carrying either a request, an acknowledgment, or a cache block of data plus some control bits. However, the basic performance model for network transactions developed in earlier chapters applies here as well. A typical network transaction incurs some overhead on the processor at its source (traversing the cache hierarchy on the way out and back in), some work or occupancy on the communication assists at its end-points (typically looking up state, generating requests, or intervening in the cache) and some delay in the network due to transit latency, network bandwidth and contention. Typically, the processor itself is not involved at the home, the dirty node, or the sharers, but only at the requestor. A protocol does not have much leverage on the basic costs of a single network transaction—transit latency, network bandwidth, assist occupancy, and processor overhead—but it can determine the number and structure of the network transactions needed to realize memory operations like reads and writes under different circumstances. In general, there are three classes of techniques for improving performance: (i) protocol optimizations, (ii) high-level machine organization, and (iii) hardware specialization to reduce latency and occupancy and increase bandwidth. The first two assume a fixed set of cost parameters for the communication architecture, and are discussed in this section. The impact of varying these basic parameters will be examined in Section 8.8, after we understand the protocols in more detail.

Protocol Optimizations

At the protocol level, the two major performance goals are: (i) to reduce the number of network transactions generated per memory operation m, which reduces the bandwidth demands placed on the network and the communication assists; and (ii) to reduce the number of actions, especially network transactions, that are on the critical path of the processor, thus reducing uncontended latency. The latter can be done by overlapping the transactions needed for a memory operation as much as possible. To some extent, protocols can also help reduce the end point assist occupancy per transaction—especially when the assists are programmable—which reduces both uncontended latency as well as end-point contention. The traffic, latency and occupancy characteristics should not scale up quickly with the number of processing nodes used, and should perform gracefully under pathological conditions like hot-spots.

As we have seen, the manner in which directory information is stored determines some basic order statistics on the number of network transactions in the critical path of a memory operation. For example, a memory-based protocol can issue invalidations in an overlapped manner from the home, while in a cache-based protocol the distributed list must be walked by network transactions to learn the identities of the sharers. However, even within a class of protocols there are many ways to improve performance.

Consider read operations in a memory-based protocol. The strict request-reply option described earlier is shown in Figure 8-12(a). The home replies to the requestor with a message containing

(a) Strict request-reply   *(a) Intervention forwarding*

(c) Reply forwarding
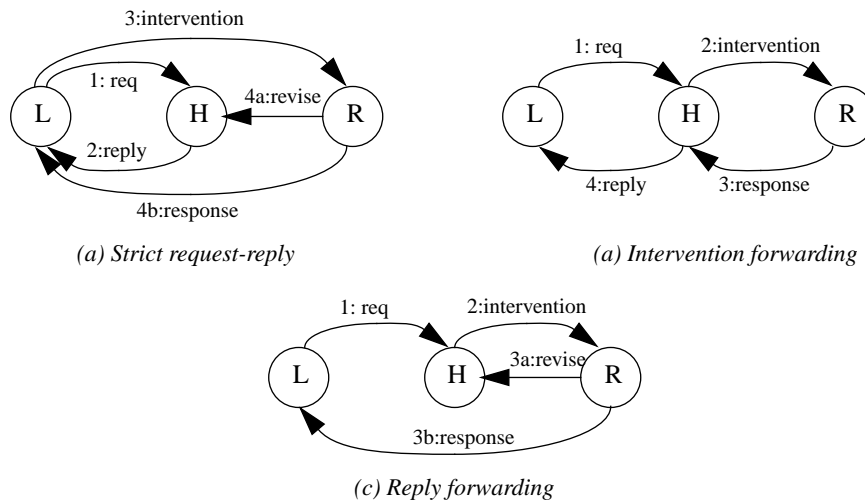
Figure 8-12 Reducing latency in a flat, memory-based protocol through forwarding.

The case shown is of a read request to a block in exclusive state. L represents the local or requesting node, H is the home for the block, and R is the remote owner node that has the exclusive copy of the block.

the identity of the owner node. The requestor then sends a request to the owner, which replies to it with the data (the owner also sends a "revision" message to the home, which updates memory with the data and sets the directory state to be shared).

There are four network transactions in the critical path for the read operation, and five transactions in all. One way to reduce these numbers is *intervention forwarding*. In this case, the home does not reply to the requestor but simply forwards the request as an *intervention* transaction to the owner, asking it to retrieve the block from its cache. An intervention is just like a request, but is issued in reaction to a request and is directed at a cache rather than a memory (it is similar to an invalidation, but also seeks data from the cache). The owner then replies to the home with the data or an acknowledgment (if the block is clean-exclusive rather than modified)—at which time the home updates its directory state and replies to the requestor with the data (Figure 8-12(b)). Intervention forwarding reduces the total number of transactions needed to four, but all four are still in the critical path. A more aggressive method is *reply forwarding* (Figure 8-12(c)). Here too, the home forwards the intervention message to the owner node, but the intervention contains the identity of the requestor and the owner replies directly to the requestor itself. The owner also sends a revision message to the home so the memory and directory can be updated. This keeps the number of transactions at four, but reduces the number in the critical path to three (request-→intervention→reply-to-requestor) and constitutes a *three-message miss*. Notice that with either

of intervention forwarding or reply forwarding the protocol is no longer strictly request-reply (a request to the home generates a request to the owner node, which generates a reply). This can complicate deadlock avoidance, as we shall see later.

Besides being only intermediate in its latency and traffic characteristics, intervention forwarding has the disadvantage that outstanding intervention requests are kept track of at the home rather than at the requestor, since responses to the interventions are sent to the home. Since requests that cause interventions may come from any of the processors, the home node must keep track of up to $k*P$ interventions at a time, where k is the number of outstanding requests allowed per processor. A requestor, on the other hand, would only have to keep track of at most k outstanding interventions. Since reply forwarding also has better performance characteristics, most systems prefer to use it. Similar forwarding techniques can be used to reduce latency in cache-based schemes at the cost of strict request-reply simplicity, as shown in Figure 8-13.

In addition to forwarding, other protocol techniques to reduce latency include overlapping transactions and activities by performing them speculatively. For example, when a request arrives at the home, the controller can read the data from memory in parallel with the directory lookup, in the hope that in most cases the block will indeed be clean at the home. If the directory lookup indicates that the block is dirty in some cache then the memory access is wasted and must be ignored. Finally, protocols may also automatically detect and adjust themselves at runtime to
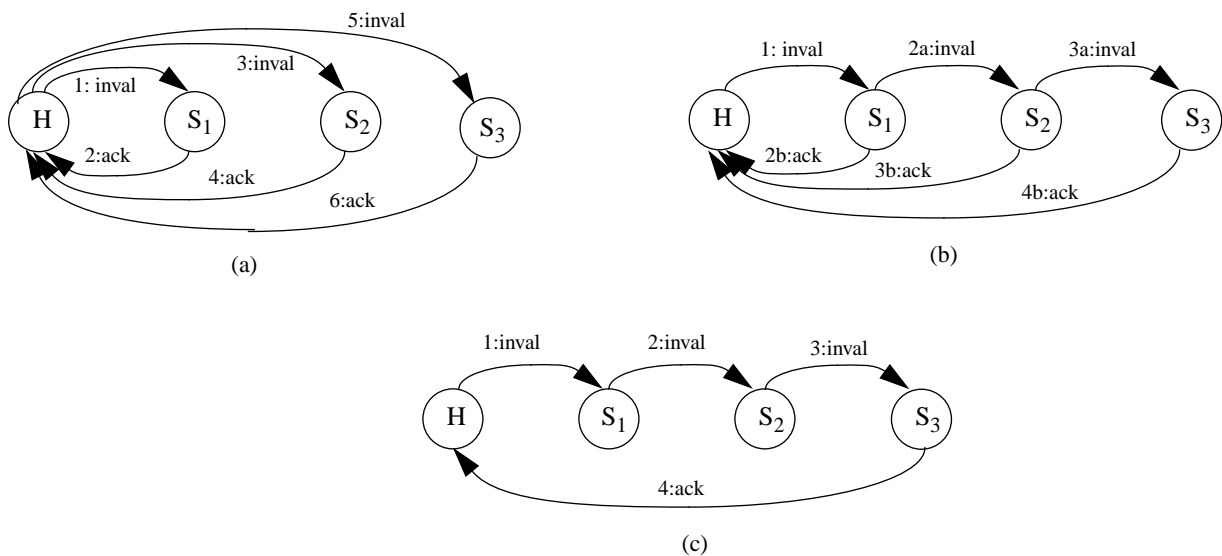
Figure 8-13 Reducing latency in a flat, cache-based protocol.

The scenario is of invalidations being sent from the home H to the sharers S_i on a write operation. In the strict request-reply case (a), every node includes in its acknowledgment (reply) the identity of the next sharer on the list, and the home then sends that sharer an invalidation. The total number of transactions in the invalidation sequence is $2s$, where s is the number of sharers, and all are in the critical path. In (b), each invalidated node forwards the invalidation to the next sharer and in parallel sends an acknowledgment to the home. The total number of transactions is still $2s$, but only $s+1$ are in the critical path. In (c), only the last sharer on the list sends a single acknowledgment telling the home that the sequence is done. The total number of transactions is $s+1$. (b) and (c) are not strict request-reply.

interact better with common sharing patterns to which the standard invalidation-based protocol is not ideally suited (see Exercise 8.4).

High-level Machine Organization

Machine organization can interact with the protocol to help improve performance as well. For example, the use of large tertiary caches can reduce the number of protocol transactions due to artifactual communication. For a fixed total number of processors, using multiprocessor rather than uniprocessor nodes may be useful as discussed in the introduction to this chapter: The directory protocol operates only across nodes, and a different coherence protocol, for example a snoopy protocol or another directory protocol, may be used to keep caches coherent within a node. Such an organization with multiprocessor—especially bus-based—nodes in a two-level hierarchy is very common, and has interesting advantages as well as disadvantages.

The potential advantages are in both cost and performance. On the cost side, certain fixed per-node costs may be amortized among the processors within a node, and it is possible to use existing SMPs which may themselves be commodity parts. On the performance side, advantages may arise from sharing characteristics that can reduce the number of accesses that must involve the directory protocol and generate network transactions across nodes. If one processor brings a block of data into its cache, another processor in the same node may be able to satisfy its miss to that block (for the same or a different word) more quickly through the local protocol, using cache-to-cache sharing, especially if the block is allocated remotely. Requests may also be combined: If one processor has a request outstanding to the directory protocol for a block, another processor's request within the same SMP can be combined with and obtain the data from the first processor's response, reducing latency, network traffic and potential hot-spot contention. These advantages are similar to those of full hierarchical approaches mentioned earlier. Within an SMP, processors may even share a cache at some level of the hierarchy, in which case the tradeoffs discussed in Chapter 6 apply. Finally, cost and performance characteristics may be improved by using a hierarchy of packaging technologies appropriately.

Of course, the extent to which the two-level sharing hierarchy can be exploited depends on the sharing patterns of applications, how well processes are mapped to processors in the hierarchy, and the cost difference between communicating within a node and across nodes. For example, applications that have wide but physically localized read-only sharing in a phase of computation, like the Barnes-Hut galaxy simulation, can benefit significantly from cache-to-cache sharing if the miss rates are high to begin with. Applications that exhibit nearest-neighbor sharing (like Ocean) can also have most of their accesses satisfied within a multiprocessor node. However, while some processes may have their all accesses satisfied within their node, others will have accesses along at least one border satisfied remotely, so there will be load imbalances and the benefits of the hierarchy will be diminished. In all-to-all communication patterns, the savings in inherent communication are more modest. Instead of communicating with *p-1* remote processors in a p-processor system, a processor now communicates with *k-1* local processors and *p-k* remote ones (where *k* is the number of processors within a node), a savings of at most *(p-k)/(p-1)*. Finally, with several processes sharing a main memory unit, it may also be easier to distribute data appropriately among processors at page granularity. Some of these tradeoffs and application characteristics are explored quantitatively in [Web93, ENS+95]. Of our two case-study machines, the NUMA-Q uses four-processor, bus-based, cache-coherent SMPs as the nodes. The SGI Origin takes an interesting position: two processors share a bus and memory (and a board) to amortize cost, but they are not kept coherent by a snoopy protocol on the bus; rather, a single directory protocol keeps all caches in the machines coherent.

Compared to using uniprocessor nodes, the major potential disadvantage of using multiprocessor nodes is bandwidth. When processors share a bus, an assist or a network interface, they amortize its cost but compete for its bandwidth. If their bandwidth demands are not reduced much by locality, the resulting contention can hurt performance. The solution is to increase the throughput of these resources as well as processors are added to the node, but this increases compromises the cost advantages. Sharing a bus within a node has some particular disadvantages. First, if the bus has to accommodate several processors it becomes longer and is not likely to be contained in a single board or other packaging unit. Both these effects slow the bus down, increasing the latency to both local and remote data. Second, if the bus supports snoopy coherence within the node, a request that must be satisfied remotely typically has to wait for local snoop results to be reported before it is sent out to the network, causing unnecessary delays. Third, with a snoopy bus at the remote node too, many references that do go remote will require snoops and data transfers on the local bus as well as the remote bus, increasing latency and reducing bandwidth. Nonetheless, several directory-based systems use snoopy-coherent multiprocessors as their individual nodes [LLJ+93, LoC96, ClA96, WGH+97].

8.5.2 Correctness

Correctness considerations can be divided into three classes. First, the protocol must ensure that the relevant blocks are invalidated/updated and retrieved as needed, and that the necessary state transitions occur. We can assume this happens in all cases, and not discuss it much further. Second, the serialization and ordering relationships defined by coherence and the consistency model must be preserved. Third, the protocol and implementation must be free from deadlock, livelock, and ideally starvation too. Several aspects of scalable protocols and systems complicate the latter two sets of issues beyond what we have seen for bus-based cache-coherent machines or scalable non-coherent machines. There are two basic problems. First, we now have multiple cached copies of a block but no single agent that sees all relevant transactions and can serialize them. Second, with many processors there may be a large number of requests directed toward a single node, accentuating the input buffer problem discussed in the previous chapter. These problems are aggravated by the high latencies in the system, which push us to exploit the protocol optimizations discussed above; these allow more transactions to be in progress simultaneously, further complicating correctness. This subsection describes the new issues that arise in each case (write serialization, sequential consistency, deadlock, livelock and starvation) and lists the major types of solutions that are commonly employed. Some specific solutions used in the case study protocols will be discussed in more detail in subsequent sections.

Serialization to a Location for Coherence

Recall the write serialization clause of coherence. Not only must a given processor be able to construct a serial order out of all the operations to a given location—at least out of all write operations and its own read operations—but all processors must see the writes to a given location as having happened in the same order.

One mechanism we need for serialization is an entity that sees the necessary memory operations to a given location from different processors (the operations that are not contained entirely within a processing node) and determines their serialization. In a bus-based system, operations from different processors are serialized by the order in which their requests appear on the bus (in fact all accesses, to any location, are serialized this way). In a distributed system that does not cache shared data, the consistent serializer for a location is the main memory that is the home of a loca-

tion. For example, the order in which writes become visible is the order in which they reach the memory, and which write's value a read sees is determined by when that read reaches the memory. In a distributed system with coherent caching, the home memory is again a likely candidate for the entity that determines serialization, at least in a flat directory scheme, since all relevant operations first come to the home. If the home could satisfy all requests itself, then it could simply process them one by one in FIFO order of arrival and determine serialization. However, with multiple copies visibility of an operation to the home does not imply visibility to all processors, and it is easy to construct scenarios where processors may see operations to a location appear to be serialized in different orders than that in which the requests reached the home, and in different orders from each other.

As a simple example of serialization problems, consider a network that does not preserve point-to-point order of transactions between end-points. If two write requests for shared data arrive at the home in one order, in an update-based protocol (for example) the updates they generate may arrive at the copies in different orders. As another example, suppose a block is in modified state in a dirty node and two nodes issues read-exclusive requests for it in an invalidation protocol. In a strict request-reply protocol, the home will provide the requestors with the identity of the dirty node, and they will send requests to it. However, with different requestors, even in a network that preserves point-to-point order there is no guarantee that the requests will reach the dirty node in the same order as they reached the home. Which entity provides the globally consistent serialization in this case, and how is this orchestrated with multiple operations for this block simultaneously in flight and potentially needing service from different nodes?

Several types of solutions can be used to ensure serialization to a location. Most of them use additional directory states called *busy states* (sometimes called *pending states* as well). A block being in busy state at the directory indicates that a previous request that came to the home for that block is still in progress and has not been completed. When a request comes to the home and finds the directory state to be busy, serialization may be provided by one of the following mechanisms.

Buffer at the home. The request may be buffered at the home as a pending request until the previous request that is in progress for the block has completed, regardless of whether the previous request was forwarded to a dirty node or a strict request-reply protocol was used (the home should of course process requests for other blocks in the meantime). This ensures that requests will be serviced in FIFO order of arrival at the home, but it reduces concurrency. The buffering must be done regardless of whether a strict request-reply protocol is used or the previous request was forwarded to a dirty node, and this method requires that the home be notified when a write has completed. More importantly, it raises the danger of the input buffer at the home overflowing, since it buffers pending requests for *all* blocks for which it is the home. One strategy in this case is to let the input buffer overflow into main memory, thus providing effectively infinite buffering as long as there is enough main memory, and thus avoid potential deadlock problems. This scheme is used in the MIT Alewife prototype [ABC+95].

Buffer at requestors. Pending requests may be buffered not at the home but at the requestors themselves, by constructing a distributed linked list of pending requests. This is clearly a natural extension of a cache-based approach which already has the support for distributed linked lists, and is used in the SCI protocol [Gus92,IEE93]. Now the number of pending requests that a node may need to keep track off is small and determined only by it.

NACK and retry. An incoming request may be NACKed by the home (i.e. negative acknowledgment sent to the requestor) rather than buffered when the directory state indicates that a previous request for the block is still in progress. The request will be retried later, and will be

serialized in the order in which it is actually accepted by the directory (attempts that are NACKed do not enter in the serialization order). This is the approach used in the Origin2000 [LaL97].

Forward to the dirty node: If a request is still outstanding at the home because it has been forwarded to a dirty node, subsequent requests for that block are not buffered or NACKed. Rather, they are forwarded to the dirty node as well, which determines their serialization. The order of serialization is thus determined by the home node when the block is clean at the home, and by the order in which requests reach the dirty node when the block is dirty. If the block leaves the dirty state before a forwarded request reaches it (for example due to a writeback or a previous forwarded request), the request may be NACKed by the dirty node and retried. It will be serialized at the home or a dirty node when the retry is successful. This approach was used in the Stanford DASH protocol [LLG+90,LLJ+93].

Unfortunately, with multiple copies in a distributed network simply identifying a serializing entity is not enough. The problem is that the home or serializing agent may know when its involvement with a request is done, but this does not mean that the request has completed with respect to other nodes. Some remaining transactions for the next request may reach other nodes and complete with respect to them before some transactions for the previous request. We will see concrete examples and solutions in our case study protocols. Essentially, we will learn that in addition to a global serializing entity for a block individual nodes (e.g. requestors) should also preserve a local serialization with respect to each block; for example, they should not apply an incoming transaction to a block while they still have a transaction outstanding for that block.

Sequential Consistency: Serialization across Locations

Recall the two most interesting components of preserving the sufficient conditions for satisfying sequential consistency: detecting write completion (needed to preserve program order), and ensuring write atomicity. In bus-based machine, we saw that the restricted nature of the interconnect allows the requestor to detect write completion early: the write can be acknowledged to the processor as soon as it obtains access to the bus, without needing to wait for it to actually invalidate or update other processors (Chapter 6). By providing a path through which all transactions pass and ensuring FIFO ordering among writes beyond that, the bus also makes write atomicity quite natural to ensure. In a machine that has a general distributed network but does not cache shared data, detecting the completion of a write requires an explicit acknowledgment from the memory that holds the location (Chapter 7). In fact, the acknowledgment can be generated early, once we know the write has reached that node and been inserted in a FIFO queue to memory; at this point, it is clear that all subsequent reads will no longer see the old value and we can assume the write has completed. Write atomicity falls out naturally: a write is visible only when it updates main memory, and at that point it is visible to all processors.

Write Completion. As mentioned in the introduction to this chapter, with multiple copies in a distributed network it is difficult to assume write completion before the invalidations or updates have actually reached all the nodes. A write cannot be acknowledged by the home once it has reached there and assume to have effectively completed. The reason is that a subsequent write Y in program order may be issued by the processor after receiving such an acknowledgment for a previous write X, but Y may become visible to another processor before X thus violating SC. This may happen because the transactions corresponding to Y took a different path through the network or if the network does not provide point-to-point order. Completion can only be assumed once explicit acknowledgments are received from all copies. Of course, a node with a copy can

generate the acknowledgment as soon as it receives the invalidation—before it is actually applied to the caches—as long as it guarantees the appropriate ordering within its cache hierarchy (just as in Chapter 6). To satisfy the sufficient conditions for SC, a processor waits after issuing a write until all acknowledgments for that write have been received, and only then proceeds past the write to a subsequent memory operation.

Write atomicity. Write atomicity is similarly difficult when there are multiple copies with no shared path to them. To see this, Figure 8-14 shows how an example from Chapter 5 (Figure 5-11) that relies on write atomicity can be violated. The constraints of sequential consistency have
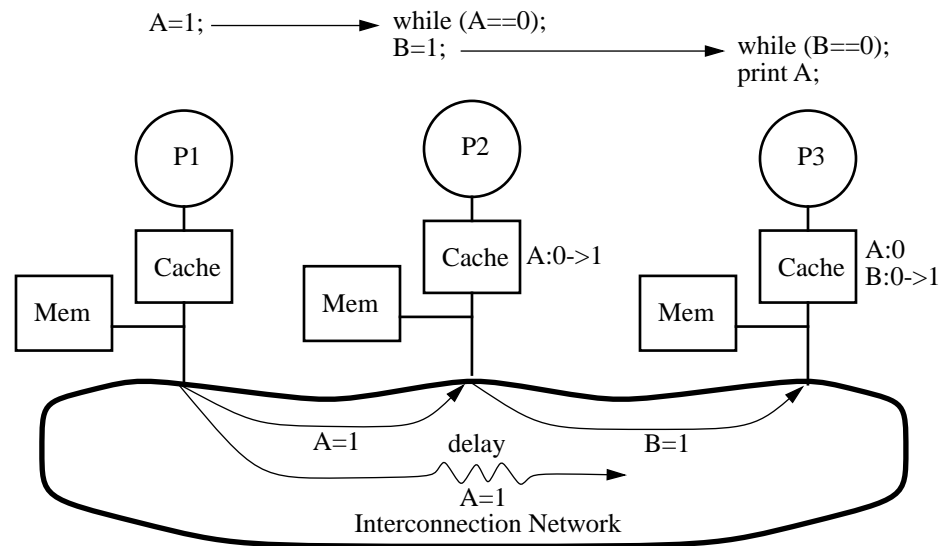
Figure 8-14 Violation of write atomicity in a scalable system with caches.

Assume that the network preserves point-to-point order, and every cache starts out with copies of A and B initialized to 0. Under SC, we expect P3 to print 1 as the value of A. However, P2 sees the new value of A, and jumps out of its while loop to write B even before it knows whether the previous write of A by P1 has become visible to P3. This write of B becomes visible to P3 before the write of A by P1, because the latter was delayed in a congested part of the network that the other transactions did not have to go through at all. Thus, P3 reads the new value of B but the old value of A.

to be satisfied by orchestrating network transactions appropriately. A common solution for write atomicity in an invalidation-based scheme is for the current owner of a block (the main-memory module or the processor holding the dirty copy in its cache) to provide the appearance of atomicity, by not allowing access to the new value by any process until all invalidation acknowledgments for the write have returned. Thus, no processor can see the new value until it is visible to all processors. Maintaining the appearance of atomicity is much more difficult for update-based protocols, since the data are sent to the readers and hence accessible immediately. Ensuring that readers do not read the value until it is visible to all of them requires a two-phase interaction. In the first-phase the copies of that memory block are updated in all processors' caches, but those processors are prohibited from accessing the new value. In the second phase, after the first phase is known to have completed through acknowledgments as above, those processors are sent messages that allow them to use the new value. This makes update protocols even less attractive for scalable directory-based machines than for bus-based machines.

Deadlock

In Chapter 7 we discussed an important source of potential deadlock in request-reply protocols such as those of a shared address space: the filling up of a finite input buffer. Three solutions were proposed for deadlock:

> **Provide enough buffer space**, either by buffering requests at the requestors using distributed linked lists, or by providing enough input buffer space (in hardware or main memory) for the maximum number of possible incoming transactions. Among cache-coherent machines, this approach is used in the MIT Alewife and the Hal S1.
>
> **Use NACKs**.
>
> **Provide separate request and reply networks**, whether physical or virtual (see Chapter 10), to prevent backups in the potentially poorly behaved request network from blocking the progress of well-behaved reply transactions.

Two separate networks would suffice in a protocol that is strictly request-reply; i.e. in which all transactions can be separated into requests and replies such that a request transaction generates only a reply (or nothing), and a reply generates no further transactions. However, we have seen that in the interest of performance many practical coherence protocols use forwarding and are not always strictly request-reply, breaking the deadlock-avoidance assumption. In general, we need as many networks (physical or virtual) as the longest chain of different transaction types needed to complete a given operation. However, using multiple networks is expensive and many of them will be under-utilized. In addition to the approaches that provide enough buffering, two different approaches can be taken to dealing with deadlock in protocols that are not strict request-reply. Both rely on detecting situations when deadlock appears possible, and resorting to a different mechanism to avoid deadlock in these cases. That mechanism may be NACKS, or reverting to a strict request-reply protocol.

The potential deadlock situations may be detected in many ways. In the Stanford DASH machine, a node conservatively assumes that deadlock may be about to happen when both its input request and output request buffers fill up beyond a threshold, and the request at the head of the input request queue is one that may need to generate further requests like interventions or invalidations (and is hence capable of causing deadlock). At this point, the node takes such requests off the input queue one by one and sends NACK messages back for them to the requestors. It does this until the request at the head is no longer one that can generate further requests, or until it finds that the output request queue is no longer full. The NACK'ed requestors will retry their requests later. An alternative detection strategy is when the output request buffer is full and has not had a transaction removed from it for T cycles.

In the other approach, potential deadlock situations are detected in much the same way. However, instead of sending a NACK to the requestor, the node sends it a reply asking it to send the intervention or invalidation requests directly itself; i.e. it dynamically backs off to a strict request-reply protocol, compromising performance temporarily but not allowing deadlock cycles. The advantage of this approach is that NACKing is a statistical rather than robust solution to such problems, since requests may have to be retried several times in bad situations. This can lead to increased network traffic and increased latency to the time the operation completes. Dynamic back-off also has advantages related to livelock, as we shall see next, and is used in the Origin2000 protocol.

Livelock

Livelock and starvation are taken care of naturally in protocols that provide enough FIFO buffering of requests, whether centralized or through distributed linked lists. In other cases, the classic livelock problem of multiple processors trying to write a block at the same time is taken care of by letting the first request to get to the home go through but NACKing all the others.

NACKs are useful mechanisms for resolving race conditions without livelock. However, when used to ease input buffering problems and avoid deadlock, as in the DASH solution above, they succeed in avoiding deadlock but have the potential to cause livelock. For example, when the node that detects a possible deadlock situation NACKs some requests, it is possible that all those requests are retried immediately. With extreme pathology, the same situation could repeat itself continually and livelock could result. (The actual DASH implementation steps around this problem by using a large enough request input buffer, since both the number of nodes and the number of possible outstanding requests per node are small. However, this is not a robust solution for larger, more aggressive machines that cannot provide enough buffer space.) The alternative solution to deadlock, of switching to a strict request-reply protocol in potential deadlock situations, does not have this problem. It guarantees forward progress and also removes the request-request dependence at the home once and for all.

Starvation

The actual occurrence of starvation is usually very unlikely in well-designed protocols; however, it is not ruled out as a possibility. The fairest solution to starvation is to buffer all requests in FIFO order; however, this can have performance disadvantages, and for protocols that do not do this avoiding starvation can be difficult to guarantee. Solutions that use NACKs and retries are often more susceptible to starvation. Starvation is most likely when many processors repeatedly compete for a resource. Some may keep succeeding, while one or more may be very unlucky in their timing and may always get NACKed.

A protocol could decide to do nothing about starvation, and rely on the variability of delays in the system to not allow such an indefinitely repeating pathological situation to occur. The DASH machine uses this solution, and times out with a bus error if the situation persists beyond a threshold time. Or a random delay can be inserted between retries to further reduce the small probability of starvation. Finally, requests may be assigned priorities based on the number of times they have been NACK'ed, a technique that is used in the Origin2000 protocol.

Having understood the basic directory organizations and high-level protocols, as well as the key performance and correctness issues in a general context, we are now ready to dive into actual case studies of memory-based and cache-based protocols. We will see what protocol states and activities look like in actual realizations, how directory protocols interact with and are influenced by the underlying processing nodes, what real scalable cache-coherent machines look like, and how actual protocols trade off performance with the complexity of maintaining correctness and of debugging or validating the protocol.

8.6 Memory-based Directory Protocols: The SGI Origin System

We discuss flat, memory-based directory protocols first, using the SGI Origin multiprocessor series as a case study. At least for moderate scale systems, this machine uses essentially a full bit vector directory representation. A similar directory representation but slightly different protocol was also used in the Stanford DASH multiprocessor [LLG+90], a research prototype which was the first distributed memory machine to incorporate directory-based coherence. We shall follow a similar discussion template for both this and the next case study (the SCI protocol as used in the Sequent NUMA-Q). We begin with the basic coherence protocol, including the directory structure, the directory and cache states, how operations such as reads, writes and writebacks are handled, and the performance enhancements used. Then, we briefly discuss the position taken on the major correctness issues, followed by some prominent protocol extensions for extra functionality. Having discussed the protocol, we describe the rest of the machine as a multiprocessor and how the coherence machinery fits into it. This includes the processing node, the interconnection network, the input/output, and any interesting interactions between the directory protocol and the underlying node. We end with some important implementation issues—illustrating how it all works and the important data and control pathways—the basic performance characteristics (latency, occupancy, bandwidth) of the protocol, and the resulting application performance for our sample applications.

8.6.1 Cache Coherence Protocol

The Origin system is composed of a number of processing nodes connected by a switch-based interconnection network (see Figure 8-15). Every processing node contains two processors, each with first- and second-level caches, a fraction of the total main memory on the machine, an I/O interface, and a single-chip communication assist or coherence controller called the Hub which implements the coherence protocol. The Hub is integrated into the memory system. It sees all cache misses issued by the processors in that node, whether they are to be satisfied locally or remotely, receives transactions coming in from the network (in fact, the Hub implements the network interface as well), and is capable of retrieving data from the local processor caches.

In terms of the performance issues discussed in Section 8.5.1, at the protocol level the Origin2000 uses reply forwarding as well as speculative memory operations in parallel with directory lookup at the home. At the machine organization level, the decision in Origin to have two processors per node is driven mostly by cost: Several other components on a node (the Hub, the system bus, etc.) are shared between the processors, thus amortizing their cost while hopefully still providing substantial bandwidth per processor. The Origin designers believed that the latency disadvantages of a snoopy bus discussed earlier outweighed its advantages (see Section 8.5.1), and chose not to maintain snoopy coherence between the two processors within a node. Rather, the bus is simply a shared physical link that is multiplexed between the two processors in a node. This sacrifices the potential advantage of cache to cache sharing within the node, but helps reduce latency. With a Hub shared between two processors, combining of requests to the network (not the protocol) could nonetheless have been supported, but is not due to the additional implementation cost. When discussing the protocol in this section, let us assume for simplicity that each node contains only one processor, its cache hierarchy, a Hub, and main memory. We will see the impact of using two processors per node on the directory structure and protocol later.
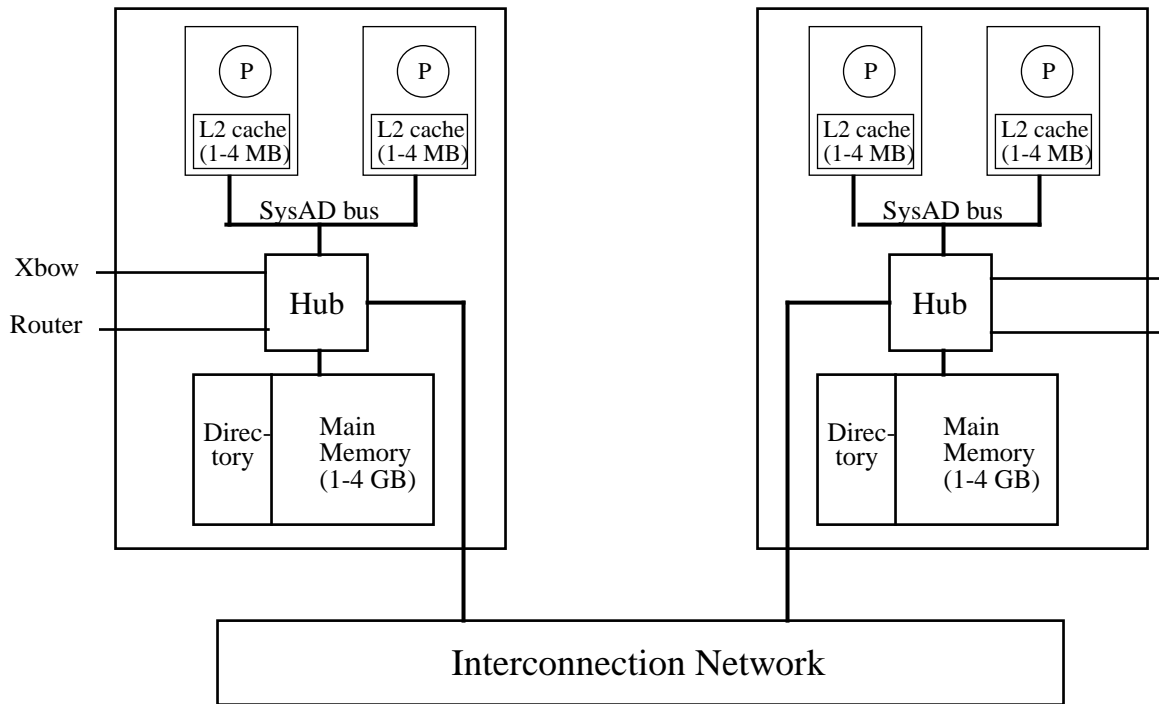
Figure 8-15 Block diagram of the Silicon Graphics Origin multiprocessor.

Other than the use of reply forwarding, the most interesting aspects of the Origin protocol are its use of busy states and NACKs to resolve race conditions and provide serialization to a location, and the way in which it handles race conditions caused by writebacks. However, to illustrate how a complete protocol works in light of these races, as well as the performance enhancement techniques used in different cases, we will describe how it deals with simple read and write requests as well.

Directory Structure and Protocol States

The directory information for a memory block is maintained at the home node for that block. How the directory structure deals with the fact that each node is two processors and how the directory organization changes with machine size are interesting aspects of the machine, and we shall discuss these after the protocol itself. For now, we can assume a full bit vector approach like in the simple protocol of Section 8.3.

In the caches, the protocol uses the states of the MESI protocol, with the same meanings as used in Chapter 5. At the directory, a block may be recorded as being in one of seven states. Three of these are basic states: (i) *Unowned*, or no copies in the system, (ii) *Shared*, i.e. zero or more read-only copies, their whereabouts indicated by the presence vector, and (iii) *Exclusive*, or one read-write copy in the system, indicated by the presence vector. An exclusive directory state means the block might be in either dirty or clean-exclusive state in the cache (i.e. the M or E states of the

MESI protocol). Three other states are *busy* states. As discussed earlier, these imply that the home has received a previous request for that block, but was not able to complete that operation itself (e.g. the block may have been dirty in a cache in another node); transactions to complete the request are still in progress in the system, so the directory at the home is not yet ready to handle a new request for that block. The three busy states correspond to three different types of requests that might still be thus in progress: a read, a read-exclusive or write, and an uncached read (discussed later). The seventh state is a *poison* state, which is used to implement a lazy TLB shootdown method for migrating pages among memories (Section 8.6.4). Given these states, let us see how the coherence protocol handles read requests, write requests and writeback requests from a node. The protocol does not rely on any order preservation among transactions in the network, not even point-to-point order among transactions between the same nodes.

Handling Read Requests

Suppose a processor issues a read that misses in its cache hierarchy. The address of the miss is examined by the local Hub to determine the home node, and a read request transaction is sent to the home node to look up the directory entry. If the home is local, it is looked up by the local Hub itself. At the home, the data for the block is accessed speculatively in parallel with looking up the directory entry. The directory entry lookup, which completes a cycle earlier than the speculative data access, may indicate that the memory block is in one of several different states—shared, unowned, busy, or exclusive—and different actions are taken in each case.

Shared or *unowned*: This means that main memory at the home has the latest copy of the data (so the speculative access was successful). If the state is shared, the bit corresponding to the requestor is set in the directory presence vector; if it is unowned, the directory state is set to exclusive. The home then simply sends the data for the block back to the requestor in a reply transaction. These cases satisfy a strict request-reply protocol. Of course, if the home node is the same as the requesting node, then no network transactions or messages are generated and it is a locally satisfied miss.

Busy: This means that the home should not handle the request at this time since a previous request has been forwarded from the home and is still in progress. The requestor is sent a *negative acknowledge* (*NACK*) message, asking it to try again later. A NACK is categorized as a reply, but like an acknowledgment it does not carry data.

Exclusive: This is the most interesting case. If the home is not the owner of the block, the valid data for the block must be obtained from the owner and must find their way to the requestor as well as to the home (since the state will change to shared). The Origin protocol uses reply forwarding to improve performance over a strict request-reply approach: The request is forwarded to the owner, which replies directly to the requestor, sending a revision message to the home. If the home itself is the owner, then the home can simply reply to the requestor, change the directory state to shared, and set the requestor's bit in the presence vector. In fact, this case is handled just as if the owner were not the home, except that the "messages" between home and owner do not translate to network transactions.

Let us look in a little more detail at what really happens when a read request arrives at the home and finds the state exclusive. (This and several other cases are also shown pictorially in Figure 8-16.) As already mentioned, the main memory block is accessed speculatively in parallel with the directory. When the directory state is discovered to be exclusive, it is set to busy-exclusive state. This is a general principle used in the coherence protocol to prevent race conditions and provide serialization of requests at the home. The state is set to busy whenever a request is received that

cannot be satisfied by the main memory at the home. This ensures that subsequent requests for that block that arrive at the home will be NACK'ed until they can be guaranteed to see the consistent final state, rather than having to be buffered at the home. For example, in the current situation we cannot set the directory state to shared yet since memory does not yet have an up to date copy, and we do not want to leave it as exclusive since then a subsequent request might chase the same exclusive copy of the block as the current request, requiring that serialization be determined by the exclusive node rather than by the home (see earlier discussion of serialization options).

Having set the directory entry to busy state, the presence vector is changed to set the requestor's bit and unset the current owner's. Why this is done at this time will become clear when we examine writeback requests. Now we see an interesting aspect of the protocol: Even though the directory state is exclusive, the home optimistically assumes that the block will be in clean-exclusive rather than dirty state in the owner's cache (so the block in main memory is valid), and sends the speculatively accessed memory block as a *speculative reply* to the requestor. At the same time, the home forwards the intervention request to the owner. The owner checks the state in its cache, and performs one of the following actions:

- If the block is dirty, it sends a response with the data directly to the requestor and also a revision message containing the data to the home. At the requestor, the response overwrites the speculative transfer that was sent by the home. The revision message with data sent to the home is called a *sharing writeback*, since it writes the data back from the owning cache to main memory and tells it to set the block to shared state.

- If the block is clean-exclusive, the response to the requestor and the revision message to the home do not contain data since both already have the latest copy (the requestor has it via the speculative reply). The response to the requestor is simply a completion acknowledgment, and the revision message is called a *downgrade* since it simply asks the home to downgrade the state of the block to shared.

In either case, when the home receives the revision message it changes the state from busy to shared.

You may have noticed that the use of speculative replies does not have any significant performance advantage in this case, since the requestor has to wait to know the real state at the exclusive node anyway before it can use the data. In fact, a simpler alternative to this scheme would be to simply assume that the block is dirty at the owner, not send a speculative reply, and always have the owner send back a response with the data regardless of whether it has the block in dirty or clean-exclusive state. Why then does Origin use speculative replies in this case? There are two reasons, which illustrate both how a protocol is influenced by the use of existing processors and their quirks as well as how different protocol optimizations influence each other. First, the R10000 processor it uses happens to not return data when it receives in intervention to a clean-exclusive (rather than dirty) block. Second, speculative replies enable a different optimization used in the Origin protocol, which is to allow a processor to simply drop a clean-exclusive block when it is replaced from the cache, rather than write it back to main memory, since main memory will in any case send a speculative reply when needed (see Section 8.6.4).

Handling Write Requests

As we saw in Chapter 5, write "misses" that invoke the protocol may generate either read-exclusive requests, which request both data and ownership, or upgrade requests which request only ownership since the requestor's data is valid. In either case, the request goes to the home, where

the directory state is looked up to determine what actions to take. If the state at the directory is anything but unowned (or busy) the copies in other caches must be invalidated, and to preserve the ordering model invalidations must be explicitly acknowledged.

As in the read case, a strict request-reply protocol, intervention forwarding, or reply forwarding can be used (see Exercise 8.2), and Origin chooses reply forwarding to reduce latency: The home updates the directory state and sends the invalidations directly; it also includes the identity of the requestor in the invalidations so they are acknowledged directly back to the requestor itself. The actual handling of the read exclusive and upgrade requests depends on the state of the directory entry when the request arrives; i.e. whether it is unowned, shared, exclusive or busy.

(a) A read or read exclusive request to a block in shared or unowned state at the directory. An exclusive reply is sent even in the read case if the block is in unowned state at the directory, so it may be loaded in E rather than S state in the cache

(b) Read or RdEx to a block in exclusive directory state. The intervention may be of type shared or exclusive, respectively (latter causes invalidation as well). The revision message is a sharing writeback or an ownership transfer, resp.

(c) Read/RdEx request to directory in busy state, or upgrade request to directory in busy, unowned or exclusive states (latter two are inappropriate)

(d) RdEx or upgrade request to directory in shared state

(e) Writeback request to directory in exclusive state

(b) WB req. to dir BUSY (the X-subscripted transactions and dashed arcs are those for the other request that made the directory busy, see text)

Figure 8-16 Pictorial depiction of protocol actions in response to requests in the Origin multiprocessor.

The case or cases under consideration in each case are shown below the diagram, in terms of the type of request and the state of the directory entry when the request arrives at the home. The messages or types of transactions are listed next to each arc. Since the same diagram represents different request type and directory state combinations, different message types are listed on each arc, separated by slashes.

Unowned: If the request is an upgrade, the state being unowned means that the block has been replaced from the requestor's cache, and the directory notified, since it sent the upgrade request (this is possible since the Origin protocol does not assume point-to-point network order). An upgrade is no longer the appropriate request, so it is NACKed. The write operation will be retried, presumably as a read-exclusive. If the request is a read-exclusive, the directory state is changed to dirty and the requestor's presence bit is set. The home replies with the data from memory.

Shared: The block must be invalidated in the caches that have copies, if any. The Hub at the home first makes a list of sharers that are to be sent invalidations, using the presence vector. It then sets the directory state to exclusive and sets the presence bit for the requestor. This ensures that subsequent requests for the block will be forwarded to the requestor. If the request was a read-exclusive, the home next sends a reply to the requestor (called an "exclusive reply with invalidations pending") which also contains the number of sharers from whom to expect invalidation acknowledgments. If the request was an upgrade, the home sends an "upgrade acknowledgment with invalidations pending" to the requestor, which is similar but does not carry the data for the block. In either case, the home next sends invalidation requests to all the sharers, which in turn send acknowledgments to the requestor (not the home). The requestor waits for all acknowledgments to come in before it "closes" or completes the operation. Now, if a new request for the block comes to the home, it will see the directory state as exclusive, and will be forwarded as an intervention to the current requestor. This current requestor will not handle the intervention immediately, but will buffer it until it has received all acknowledgments for its own request and closed the operation.

Exclusive: If the request is an upgrade, then an exclusive directory state means another write has beaten this request to the home and changed the value of the block in the meantime. An upgrade is no longer the appropriate request, and is NACK'ed. For read-exclusive requests, the following actions are taken. As with reads, the home sets the directory to a busy state, sets the presence bit of the requestor, and sends a speculative reply to it. An invalidation request is sent to the owner, containing the identity of the requestor (if the home is the owner, this is just an intervention in the local cache and not a network transaction). If the owner has the block in dirty state, it sends a "transfer of ownership" revision message to the home (no data) and a response with the data to the requestor. This response overrides the speculative reply that the requestor receives from the home. If the owner has the block in clean-exclusive state, it relies on the speculative reply from the home and simply sends an acknowledgment to the requestor and a "transfer of ownership" revision to the home.

Busy: the request is NACK'ed as in the read case, and must try again.

Handling Writeback Requests and Replacements

When a node replaces a block that is dirty in its cache, it generates a writeback request. This request carries data, and is replied to with an acknowledgment by the home. The directory cannot be in unowned or shared state when a writeback request arrives, because the writeback requestor has a dirty copy (a read request cannot change the directory state to shared in between the generation of the writeback and its arrival at the home to see a shared or unowned state, since such a request would have been forwarded to the very node that is requesting the writeback and the directory state would have been set to busy). Let us see what happens when the writeback request reaches the home for the two possible directory states: exclusive and busy.

Exclusive: The directory state transitions from exclusive to unowned (since the only cached copy has been replaced from its cache), and an acknowledgment is returned.

Busy: This is an interesting race condition. The directory state can only be busy because an intervention for the block (due to a request from another node Y, say) has been forwarded to the very node X that is doing the writeback. The intervention and writeback have crossed each other. Now we are in a funny situation. The other operation from Y is already in progress and cannot be undone. We cannot let the writeback be dropped, or we would lose the only valid copy of the block. Nor can we NACK the writeback and retry it after the operation from Y completes, since then Y's cache will have a valid copy while a different dirty copy is being written back to memory from X's cache! The protocol solves this problem by essentially combining the two operations. The writeback that finds the directory state busy changes the state to either shared (if the state was busy-shared, i.e. the request from Y was for a read copy) or exclusive (if it was busy-exclusive). The data returned in the writeback is then forwarded by the home to the requestor Y. This serves as the response to Y, instead of the response it would have received directly from X if there were no writeback. When X receives an intervention for the block, it simply ignores it (see Exercise 8.5). The directory also sends a writeback-acknowledgment to X. Node Y's operation is complete when it receives the response, and the writeback is complete when X receives the writeback-acknowledgment. We will see an exception to this treatment in a more complex case when we discuss the serialization of operations. In general, writebacks introduce many subtle situations into coherence protocols.

If the block being replaced from a cache is in shared state, we mentioned in Section 8.3 that the node may or may not choose to send a *replacement hint* message back to the home, asking the home to clear its presence bit in the directory. Replacement hints avoid the next invalidation to that block, but they incur assist occupancy and do not reduce traffic. In fact, if the block is not written again by a different node then the replacement hint is a waste. The Origin protocol chooses not to use replacement hints.

In all, the number of transaction types for coherent memory operations in the Origin protocol is 9 requests, 6 invalidations and interventions, and 39 replies. For non-coherent operations such as uncached memory operations, I/O operations and special synchronization support, the number of transactions is 19 requests and 14 replies (no invalidations or interventions).

8.6.2 Dealing with Correctness Issues

So far we have seen what happens at different nodes upon read and write misses, and how some race conditions—for example between requests and writebacks—are resolved. Let us now take a different cut through the Origin protocol, examining how it addresses the correctness issues discussed in Section 8.5.2 and what features the machine provides to deal with errors that occur.

Serialization to a Location for Coherence

The entity designated to serialize cache misses from different processors is the home. Serialization is provided not by buffering requests at the home until previous ones have completed or by forwarding them to the owner node even when the directory is in a busy state, but by NACKing requests from the home when the state is busy and causing them to be retried. Serialization is determined by the order in which the home *accepts* the requests—i.e. satisfies them itself or forwards them—not the order in which they first arrive at the home.

The earlier general discussion of serialization suggested that more was needed for coherence on a given location than simply a global serializing entity, since the serializing entity does not have

full knowledge of when transactions related to a given operation are completed at all the nodes involved. Having understood a protocol in some depth, we are now ready examine some concrete examples of this second problem [Len92] and see how it might be addressed.

Example 8-1   Consider the following simple piece of code. The write of A may happen either

P1	P2
rd A (i)	wr A
BARRIER	BARRIER
rd A (ii)	

before the first read of A or after it, but should be serializable with respect to it. The second read of A should in any case return the value written by P2. However, it is quite possible for the effect of the write to get lost if we are not careful. Show how this might happen in a protocol like the Origin's, and discuss possible solutions.

Answer   Figure 8-17 shows how the problem can occur, with the text in the figure explaining
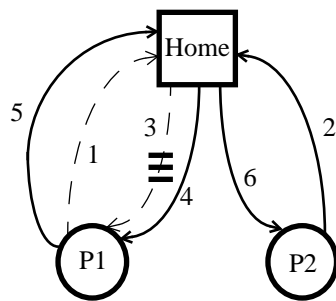
1. P1 issues read request to home node for A

2. P2 issues read-excl. request to home (for the write of A). Home (serializer) won't process it until it is done with read

3. Home receives 1, and in response sends reply to P1 (and sets directory presence bit). Home now thinks read is complete (there are no acknowledgments for a read reply). Unfortunately, the reply does not get to P1 right away.

4. In response to 2, home sends invalidate to P1; it reaches P1 before transaction 3 (there is no point-to-point order assumed in Origin, and in any case the invalidate is a request and 3 is a reply).

5. P1 receives and applies invalidate, sends ack to home.

6. Home sends data reply to P2 corresponding to request 2.

Finally, the read reply (3) reaches P1, and overwrites the invalidated block.

When P1 reads A after the barrier, it reads this old value rather than seeing an invalid block and fetching the new value. The effect of the write by P2 is lost as far as P1 is concerned.

Figure 8-17 Example showing how a write can be lost even though home thinks it is doing things in order.

Transactions associated with the first read operation are shown with dotted lines, and those associated with the write operation are shown in solid lines. Three solid bars going through a transaction indicates that it is delayed in the network.

the transactions, the sequence of events, and the problem. There are two possible solutions. An unattractive one is to have read replies themselves be acknowledged explicitly, and let the home go on to the next request only after it receives this acknowledgment. This causes the same buffering and deadlock problems as before,

further violates the request-reply nature of the protocol, and leads to long delays. The more likely solution is to ensure that a node that has a request outstanding for a block, such as P1, does not allow access by another request, such as the invalidation, to that block in its cache until its outstanding request completes. P1 may buffer the invalidation request, and apply it only after the read reply is received and completed. Or P1 can apply the invalidation even before the reply is received, and then consider the read reply invalid (a NACK) when it returns and retry it. Origin uses the former solution, while the latter is used in DASH. The buffering needed in Origin is only for replies to requests that this processor generates, so it is small and does not cause deadlock problems.

Example 8-2   Not only the requestor, but also the home may have to disallow new operations to be actually applied to a block before previous ones have completed as far as it is concerned. Otherwise directory information may be corrupted. Show an example illustrating this need and discuss solutions.

Answer   This example is more subtle and is shown in Figure 8-18. The node issuing the write request detects completion of the write (as far is its involvement is concerned) through acknowledgments before processing another request for the block. The problem is that the home does not wait for its part of the write operation—which includes waiting for the revision message and directory update—to complete before it allows another access (here the writeback) to be applied to the block. The Origin
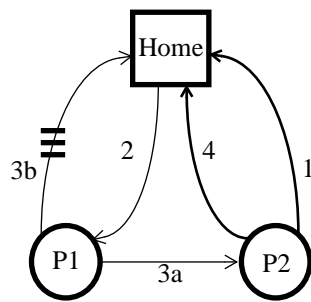
Initial Condition: block is in dirty state in P1's cache

1. P2 sends read-exclusive request to home.

2. Home forwards request to P1 (dirty node).

3. P1 sends data reply to P2 (3a), and "ownership-transfer" revise message to home to change owner to P2 (3b).

4. P2, having received its reply, considers write complete. Proceeds, but incurs a replacement of the just-dirtied block, causing it to be written back in transaction 4.

This writeback is received by the home before the ownership transfer request from P1 (even point-to-point network order wouldn't help), and the block is written into memory. Then, when the revise message arrives at the home, the directory is made to point to P2 as having the dirty copy. But this is untrue, and our protocol is corrupted.

Figure 8-18 Example showing how directory information can be corrupted if a node does not wait for a previous request to be responded to before it allows a new access to the same block.

protocol solves this problem through its busy state: The directory will be in busy-exclusive state when the writeback arrives before the revision message. When the directory detects that the writeback is coming from the same node that put the directory into busy-exclusive state, the writeback is NACKed and must be retried. (Recall from the discussion of handling writebacks that the writeback was treated

> differently if the request that set the state to busy came from a different node than the one doing the writeback; in that case the writeback was not NACK'ed but sent on as the reply to the requestor.)

These examples illustrate the importance of another general requirement that nodes must locally fulfil for proper serialization, beyond the existence of a serializing entity: Any node, not just the serializing entity, should not apply a transaction corresponding to a new memory operation to a block until a previously outstanding memory operation on that block is complete as far as that node's involvement is concerned.

Preserving the Memory Consistency Model

The dynamically scheduled R10000 processor allows independent memory operations to issue out of program order, allowing multiple operations to be outstanding at a time and obtaining some overlap among them. However, it ensures that operations complete in program order, and in fact that writes leave the processor environment and become visible to the memory system in program order with respect to other operations, thus preserving sequential consistency (Chapter 11 will discuss the mechanisms further). The processor does not satisfy the sufficient conditions for sequential consistency spelled out in Chapter 5, in that it does not wait to issue the next operation till the previous one completes, but it satisfies the model itself.[1]

Since the processor guarantees SC, the extended memory hierarchy can perform any reorderings to different locations that it desires without violating SC. However, there is one implementation consideration that is important in maintaining SC, which is due to the Origin protocol's interactions with the processor. Recall from Figure 8-16(d) what happens on a write request (read exclusive or upgrade) to a block in shared state. The requestor receives two types of replies: (i) an *exclusive reply* from the home, discussed earlier, whose role really is to indicate that the write has been serialized at memory with respect to other operations for the block, and (ii) *invalidation acknowledgments* indicating that the other copies have been invalidated and the write has completed. The microprocessor, however, expects only a single reply to its write request, as in a uniprocessor system, so these different replies have to be dealt with by the requesting Hub. To ensure sequential consistency, the Hub must pass the reply on to the processor—allowing it to declare completion—only when both the exclusive reply and the invalidation acknowledgments have been received. It must not pass on the reply simply when the exclusive reply has been received, since that would allow the processor to complete later accesses to other locations even before all invalidations for this one have been acknowledged, violating sequential consistency. We will see in Section 9.2 that such violations are useful when more relaxed memory consistency models than sequential consistency are used. Write atomicity is provided as discussed earlier: A node does not allow any incoming accesses to a block for which invalidations are outstanding until the acknowledgments for those invalidations have returned.

---

1. This is true for accesses that are under the control of the coherence protocol. The processor also supports memory operations that are not visible to the coherence protocol, called non-coherent memory operations, for which the system does not guarantee any ordering: it is the user's responsibility to insert synchronization to preserve a desired ordering in these cases.

Deadlock, Livelock and Starvation

The Origin uses finite input buffers and a protocol that is not strict request-reply. To avoid dead-lock, it uses the technique of reverting to a strict request-reply protocol when it detects a high contention situation that may cause deadlock, as discussed in Section 8.5.2. Since NACKs are not used to alleviate contention, livelock is avoided in these situations too. The classic livelock prob-lem due to multiple processors trying to write a block at the same time is avoided by using the busy states and NACKs. The first of these requests to get to the home sets the state to busy and makes forward progress, while others are NACK'ed and must retry.

In general, the philosophy of the Origin protocol is two-fold: (i) to be "memory-less", i.e. every node reacts to incoming events using only local state and no history of previous events; and (ii) to not have an operation hold globally shared resources while it is requesting other resources. The latter leads to the choices of NACKing rather than buffering for a busy resource, and helps pre-vent deadlock. These decisions greatly simplify the hardware, yet provide high performance in most cases. However, since NACKs are used rather than FIFO ordering, there is still the problem of starvation. This is addressed by associating priorities with requests. The priority is a function of the number of times the request has been NACK'ed, so starvation should be avoided.[1]

Error Handling

Despite a correct protocol, hardware and software errors can occur at runtime. These can corrupt memory or write data to different locations than expected (e.g. if the address on which to perform a write becomes corrupted). The Origin system provides many standard mechanisms to handle hardware errors on components. All caches and memories are protected by error correction codes (ECC), and all router and I/O links protected by cyclic redundancy checks (CRC) and a hardware link-level protocol that automatically detects and retries failures. In addition, the system provides mechanisms to contain failures within the part of the machine in which the program that caused the failure is running. Access protection rights are provided on both memory and I/O devices, preventing unauthorized nodes from making modifications. These access rights also allow the operating system to be structured into cells or partitions. A cell is a number of nodes, configured at boot time. If an application runs within a cell, it may be disallowed from writing memory or I/O outside that cell. If the application fails and corrupts memory or I/O, it can only affect other applications or the system running within that cell, and cannot harm code running in other cells. Thus, a cell is the unit of fault containment in the system.

---

1. The priority mechanism works as follows. The directory entry for a block has a "current" priority associated with it. Incoming transactions that will not cause the directory state to become busy are always serviced. Other transactions are only potentially serviced if their priority is greater than or equal to the current directory priority. If such a transaction is NACK'ed (e.g. because the directory is in busy state when it arrives), the current priority of the directory is set to be equal to that of the NACK'ed request. This ensures that the directory will no longer service another request of lower priority until this one is serviced upon retry. To prevent a monotonic increase and "topping out" of the directory entry priority, it is reset to zero whenever a request of priority greater than or equal to it is serviced.

8.6.3 Details of Directory Structure

While we assumed a full bit-vector directory organization so far for simplicity, the actual structure is a little more complex for two reasons: first, to deal with the fact that there are two processors per node, and second, to allow the directory structure to scale to more than 64 nodes. Bits in the bit vector usually correspond to nodes, not individual processors. However, there are in fact three possible formats or interpretations of the directory bits. If a block is in an exclusive state in a processor cache, then the rest of the directory entry is not a bit vector with one bit on but rather contains an explicit pointer to that specific processor, not node. This means that interventions forwarded from the home are targeted to a specific processor. Otherwise, for example if the block is cached in shared state, the directory entry is interpreted as a bit vector. There are two directory entry sizes: 16-bit and 64-bit (in the 16-bit case, the directory entry is stored in the same DRAM as the main memory, while in the 64-bit case the rest of the bits are in an extended directory memory module that is looked up in parallel). Even though the processors within a node are not cache-coherent, the unit of visibility to the directory in this format is a node or Hub, not a processor as in the case where the block is in an exclusive state. If an invalidation is sent to a Hub, unlike an intervention it is broadcast to both processors in the node over the SysAD bus. The 16-bit vector therefore supports up to 32 processors, and the 64-bit vector supports up to 128 processors.

For larger systems, the interpretation of the bits changes to the third format. In a p-node system, each bit now corresponds to $p/64$ nodes. The bit is set when any one (or more) of the nodes in that group of $p/64$ has a copy of the block. If a bit is set when a write happens, then invalidations are broadcast to all the $p/64$ nodes represented by that bit. For example, with 1024 processors (512 nodes) each bit corresponds to 8 nodes. This is called a *coarse vector* representation, and we shall see it again when we discuss overflow strategies for directory representations as an advanced topic in Section 8.11. In fact, the system dynamically chooses between the bit vector and coarse vector representation on a large machine: if all the nodes sharing the block are within the same 64-node "octant" of the machine, a bit-vector representation is used; otherwise a coarse vector is used.

8.6.4 Protocol Extensions

In addition to the protocol optimizations such as reply forwarding speculative memory accesses the Origin protocol provides some extensions to support special operations and activities that interact with the protocol. These include input/output and DMA operations, page migration and synchronization.

Support for Input/Output and DMA Operations

To support reads by a DMA device, the protocol provides "uncached read shared" requests. Such a request returns to the DMA device a snapshot of a coherent copy of the data, but that copy is then no longer kept coherent by the protocol. It is used primarily by the I/O system and the block transfer engine, and as such is intended for use by the operating system. For writes from a DMA device, the protocol provides "write-invalidate" requests. A write invalidate simply blasts the new value into memory, overwriting the previous value. It also invalidates all existing cached copies of the block in the system, thus returning the directory entry to unowned state. From a protocol perspective, it behaves much like a read-exclusive request, except that it modifies the block in memory and leaves the directory in unowned state.

Support for Automatic Page Migration

As we discussed in Chapter 3, on a machine with physically distributed memory it is often important for performance to allocate pages of data intelligently across physical memories, so that most capacity, conflict and cold misses are satisfied locally. Despite the very aggressive communication architecture in the Origin (much more so than other existing machines) the latency of an access satisfied by remote memory is at least 2-3 times that of a local access even without contention. The appropriate distribution of pages among memories might change dynamically at runtime, either because a parallel application's access patterns change or because the operating system decides to migrate an application process from one processor to another for better resource management across multiprogrammed applications. It is therefore useful for the system to detect the need for moving pages at runtime, and migrate them automatically to where they are needed.

Origin provides a miss counter per node at the directory for every page, to help determine when most of the misses to a page are coming from a nonlocal processor so that the page should be migrated. When a request comes in for a page, the miss counter for that node is incremented and compared with the miss counter for the home node. If it exceeds the latter by more than a threshold, then the page can be migrated to that remote node. (Sixty four counters are provided per page, and in a system with more than 64 nodes eight nodes share a counter.) Unfortunately, page migration is typically very expensive, which often annuls the advantage of doing the migration. The major reason for the high cost is not so much moving the page (which with the block transfer engine takes about 25-30 microseconds for a 16KB page) as changing the virtual to physical mappings in the TLBs of all processors that have referenced the page. Migrating a page keeps the virtual address the same but changes the physical address, so the old mappings in the page tables of processes are now invalid. As page table entries are changed, it is important that the cached versions of those entries in the TLBs of processors be invalidated. In fact, all processors must be sent a TLB invalidate message, since we don't know which ones have a mapping for the page cached in their TLB; the processors are interrupted, and the invalidating processor has to wait for the last among them to respond before it can update the page table entry and continue. This process typically takes over a hundred microseconds, in addition to the cost to move the page itself.

To reduce this cost, Origin uses a distributed lazy TLB invalidation mechanism supported by its seventh directory state, the poisoned state. The idea is to not invalidate all TLB entries when the page is moved, but rather to invalidate a processor's TLB entry only when that processor next references the page. Not only is the 100 or more microseconds of time removed from the single migrating processor's critical path, but TLB entries end up being invalidated for only those processors that actually subsequently reference the page. Let's see how this works. To migrate a page, the block transfer engine reads cache blocks from the source page location using special "uncached read exclusive" requests. This request type returns the latest coherent copy of the data and invalidates any existing cached copies (like a regular read-exclusive request), but also causes main memory to be updated with the latest version of the block and puts the directory in the poisoned state. The migration itself thus takes only the time to do this block transfer. When a processor next tries to access a block from the old page, it will miss in the cache, and will find the block in poisoned state at the directory. At that time, the poisoned state will cause the processor to see a bus error. The special OS handler for this bus error invalidates the processor's TLB entry, so that it obtains the new mapping from the page table when it retries the access. Of course, the old physical page must be reclaimed by the system at some point, to avoid wasting storage. Once the

block copy has completed, the OS invalidates one TLB entry per scheduler quantum, so that after some fixed amount of time the old page can be moved on to the free list.

Support for Synchronization

Origin provides two types of support for synchronization. First, the load linked store conditional instructions of the R10000 processor are available to compose synchronization operations, as we saw in the previous two chapters. Second, for situations in which many processors contend to update a location, such as a global counter or a barrier, uncached fetch&op primitives are provided. These fetch&op operations are performed at the main memory, so the block is not replicated in the caches and successive nodes trying to update the location do not have to retrieve it from the previous writer's cache. The cacheable LL/SC is better when the same node tends to repeatedly update the shared variable, and the uncached fetch&op is better when different nodes tend to update in an interleaved or contended way.

The protocol discussion above has provided us with a fairly complete picture of how a flat memory-based directory protocol is implemented out of network transactions and state transitions, just as a bus-based protocol was implemented out of bus transactions and state transitions. Let us now turn our attention to the actual hardware of the Origin2000 machine that implements this protocol. We first provide an overview of the system hardware organization. Then, we examine more deeply how the Hub controller is actually implemented. Finally, we will discuss the performance of the machine. Readers interested in only the protocol can skip the rest of this section, especially the Hub implementation which is quite detailed, without loss of continuity.

8.6.5 Overview of Origin2000 Hardware

As discussed earlier, each node of the Origin2000 contains two MIPS R10000 processors connected by a system bus, a fraction of the main memory on the machine (1-4 GB per node), the Hub which is the combined communication/coherence controller and network interface, and an I/O interface called the Xbow. All but the Xbow are on a single 16"-by-11" printed circuit board. Each processor in a node has its own separate L1 and L2 caches, with the L2 cache configurable from 1 to 4 MB with a cache block size of 128 bytes and 2-way set associativity. The directory state information is stored in the same DRAM modules as main memory or in separate modules, depending on the size of the machine, and looked up in parallel with memory accesses. There is one directory entry per main memory block. Memory is interleaved from 4 ways to 32 ways, depending on the number of modules plugged in (4-way interleaving at 4KB granularity within a module, and up to 32-way at 512MB granularity across modules). The system has up to 512 such nodes, i.e. up to 1024 R10000 processors. With a 195 MHz R10000 processor, the peak performance per processor is 390 MFLOPS or 780 MIPS (four instructions per cycle), leading to an aggregate peak performance of almost 500 GFLOPS in the machine. The peak bandwidth of the SysAD (system address and data) bus that connects the two processors is 780 MB/s, as is that of the Hub's connection to memory. Memory bandwidth itself for data is about 670MB/s. The Hub connections to the off-board network router chip and Xbow are 1.56 GB/sec each, using the same link technology. A more detailed picture of the node board is shown in Figure 8-19.

The Hub chip is the heart of the machine. It sits on the system bus of the node, and connects together the processors, local memory, network interface and Xbow, which communicate with one another through it. All cache misses, whether to local or remote memory, go through it, and it implements the coherence protocol. It is a highly integrated, 500K gate standard-cell design in

Connections to Backplane
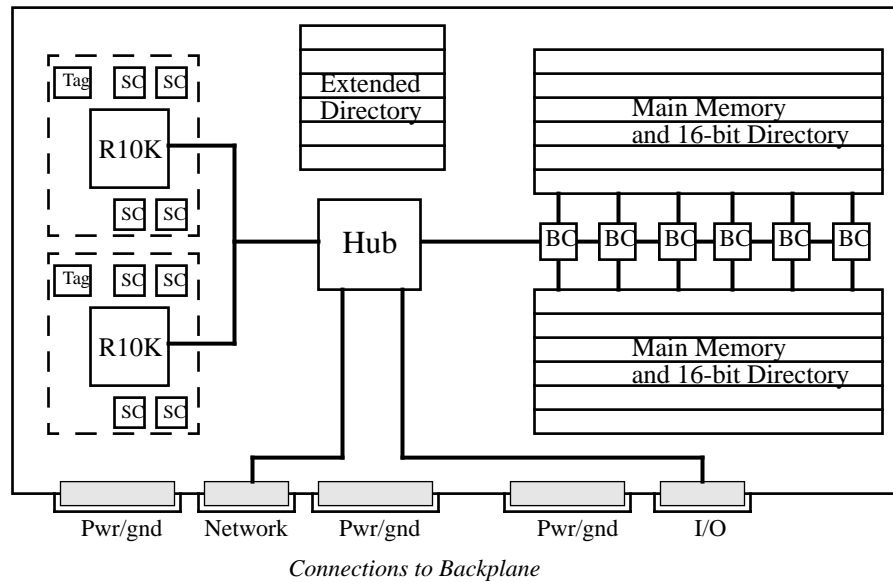
Figure 8-19 A node board on the Origin multiprocessor.

SC stands for secondary cache and BC for memory bank controller.

0.5μ CMOS technology. It contains outstanding transaction buffers for each of its two processors (each processor itself allows four outstanding requests), a pair of block transfer engines that support block memory copy and fill operations at full system bus bandwidth, and interfaces to the network, the SysAD bus, the memory/directory, and the I/O subsystem. It also implements the at-memory, uncached fetch&op instructions and page migration support discussed earlier.

The interconnection network has a hypercube topology for machines with up to 64 processors, but a different topology called a *fat cube* beyond that. This topology will be discussed in Chapter 10. The network links have very high bandwidth (1.56GB/s total per link in the two directions) and low latency (41ns pin to pin through a router), and can use flexible cabling up to 3 feet long for the links. Each link supports four *virtual channels*. Virtual channels are described in Chapter 10; for now, we can think of the machine as having four distinct networks such that each has about one-fourth of the link bandwidth. One of these virtual channels is reserved for request network transactions, one for replies. Two can be used for congestion relief and high priority transactions, thereby violating point-to-point order, or can be reserved for I/O as is usually done.

The Xbow chip connects the Hub to I/O interfaces. It is itself implemented as a cross-bar with eight ports. Typically, two nodes (Hubs) might be connected to one Xbow, and through it to six external I/O cards as shown in Figure 8-20. The Xbow is quite similar to the SPIDER router chip, but with simpler buffering and arbitration that allow eight ports to fit on the chip rather than six. It provides two virtual channels per physical channel, and supports wormhole or pipelined routing of packets. The arbiter also supports the reservation of bandwidth for certain devices, to support real time needs like video I/O. High-performance I/O cards like graphics connect directly to the Xbow ports, but most other ports are connected through a bridge to an I/O bus that allows multiple cards to plug in. Any processor can reference any physical I/O device in the machine, either through uncached references to a special I/O address space or through coherent DMA operations. An I/O device can DMA to and from any memory in the system, not just that on a

node to which it is directly connected through the Xbow. Communication between the processor and the appropriate Xbow is handled transparently by the Hubs and routers. Thus, like memory I/O is also globally accessible but physically distributed, so locality in I/O distribution is also a performance rather than correctness issue.
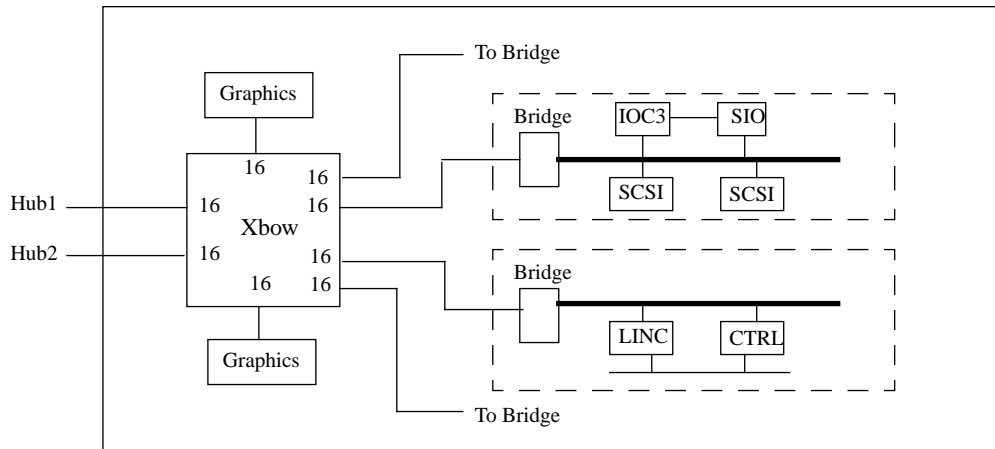
Figure 8-20 Typical Origin I/O configuration shared by two nodes.

8.6.6 Hub Implementation

The communication assist, the Hub, must have certain basic abilities to implement the protocol. It must be able to observe all cache misses, synchronization events and uncached operations, keep track of outgoing requests while moving on to other outgoing and incoming transactions, guarantee the sinking of replies coming in from the network, invalidate cache blocks, and intervene in the caches to retrieve data. It must also coordinate the activities and dependences of all the different types of transactions that flow through it from different components, and implement the necessary pathways and control. This subsection briefly describes the major components of the Hub controller used in the Origin2000, and points out some of the salient features it provides to implement the protocol. Further details of the actual data and control pathways through the Hub, as well as the mechanisms used to actually control the interactions among messages, are also very interesting to truly understand how scalable cache coherence is implemented, and can be read elsewhere [Sin97].

Being the heart of the communication architecture and coherence protocol, the Hub is a complex chip.It is divided into four major interfaces, one for each type of external entity that it connects together: the processor interface or PI, the memory/directory interface or MI, the network interface or NI, and the I/O interface (see Figure 8-21). These interfaces communicate with one another through an on-chip crossbar switch. Each interface is divided into a few major structures, including FIFO queues to buffer messages to/from other interfaces and to/from external entities. A key property of the interface design is for the interface to shield its external entity from the details of other interfaces and entities and vice versa. For example, the PI hides the processors from the rest of the world, so any other interface must only know the behavior of the PI and not
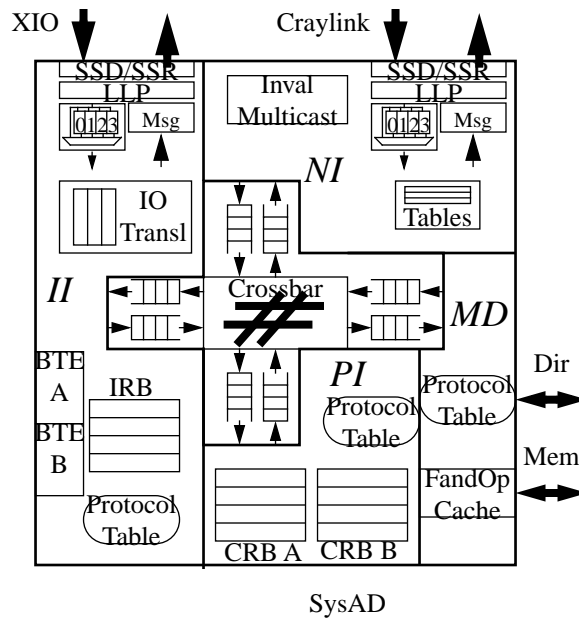
Figure 8-21 Layout of the Hub chip.

The crossbar at the center connects the buffers of the four different interfaces. Clockwise from the bottom left, the BTEs are the block transfer engines. The top left corner is the I/O interface (the SSD and SSR interface and translate signals from the I/O ports). Next is the network interface (NI), including the routing tables. The bottom right is the memory/directory (MD) interface, and at the bottom is the processor interface (PI) with its request tracking buffers (CRB and IRB).

of the processors themselves. Let us discuss the structures of the PI, MI, and NI briefly, as well as some examples of the shielding provided by the interfaces.

The Processor Interface (PI)

The PI has the most complex control mechanisms of the interfaces, since it keeps track of outstanding protocol requests and replies. The PI interfaces with the memory buses of the two R10000 processors on one side, and with incoming and outgoing FIFO queues connecting it to each of the other HUB interfaces on the other side (Figure 8-21). Each physical FIFO is logically separated into independent request and reply "virtual FIFOs" by separate logic and staging buffers. In addition, the PI contains three pairs of coherence control buffers that keep track of outstanding transactions, control the flow of messages through the PI, and implement the interactions among messages dictated by the protocol. They do not however hold the messages themselves. There are two Read Request Buffers (RRB) that track outstanding read requests from each processor, two Write Request Buffers (WRB) that track outstanding write requests, and two Intervention Request Buffers (IRB) that track incoming invalidation and intervention requests. Access to the three sets of buffers is through a single bus, so all messages contend for access to them.

A message that is recorded in one type of buffer may also need to look up another type to check for outstanding conflicting accesses or interventions to the same address (as per the coherence protocol). For example, an outgoing read request performs an associative lookup in the WRB to

see if a writeback to the same address is pending as well. If there is a conflicting WRB entry, an outbound request is not placed in the outgoing request FIFO; rather, a bit is set in the RRB entry to indicate that when the WRB entry is released the read request should be re-issued (i.e. when the writeback is acknowledged or is cancelled by an incoming invalidation as per the protocol). Buffers are also looked up to close an outstanding transaction in them when a completion reply comes in from either the processors or another interface. Since the order of completion is not deterministic, a new transaction must go into any available slot, so the buffers are implemented as fully associative rather than FIFO buffers (the queues that hold the actual messages are FIFO). The buffer lookups determine whether an external request should be generated to either the processor or the other interfaces.

The PI provides some good examples of the shielding provided by interfaces. If the processor provides data as a reply to an incoming intervention, it is the logic in the PI's outgoing FIFO that expands the reply into two replies, one to the home as a sharing writeback revision message and one to the requestor. The processor itself does not have to modified for this purpose. Another example is in the mechanisms used to keep track of and match incoming and outgoing requests and responses. All requests passing through the PI are given request numbers, and responses carry these request numbers as well. However, the processor itself does not know about these request numbers, and it is the PI's job to ensure that when it passes on incoming requests (interventions or invalidations) to the processor, that it can match the processor's replies to the outstanding interventions/invalidations without the processor having to deal with request numbers.

The Memory/Directory Interface (MI)

The MI also has FIFOs between it and the Hub crossbar. The FIFO from the Hub to the MI separates out headers from data, so that the header of the next message can be examined by the directory while the current one is being serviced, allowing writes to be pipelined and performed at peak memory bandwidth. The MI also contains a directory interface, the memory interface and t a controller. The directory interface contains the logic and tables that implement the coherence protocol as well as the logic that generates outgoing message headers; the memory interface contains the logic that generates outgoing message data. Both the memory and directory RAMS have their own address and data buses. Some messages, like revision messages coming to the home, may not access the memory but only the directory.

On a read request, the read is issued to memory speculatively simultaneously with starting the directory operation. The directory state is available a cycle before the memory data, and the controller uses this (plus the message type and initiator) to look up the directory protocol table. This hard-wired table directs it to the action to be taken and the message to send. The directory block sends the latter information to the memory block, where the message headers are assembled and inserted into the outgoing FIFO, together with the data returning from memory. The directory lookup itself is a read-modify-write operation. For this, the system provides support for partial writes of memory blocks, and a one-entry merge buffer to hold the bytes that are to be written between the time they are read from memory and written back. Finally, to speed up the at-memory fetch&op accesses provided for synchronization, the MI contains a four-entry LRU fetch&op cache to hold the recent fetch&op addresses and avoid a memory or directory access. This reduces the best-case serialization time at memory for fetch&op to 41ns, about four 100MHz Hub cycles.

The Network Interface (NI)

The NI interfaces the Hub crossbar to the network router for that node. The router and the Hub internals use different data transport formats, protocols and speeds (100MHz in the Hub versus 400MHz in the router), so one major function of the NI is to translate between the two. Toward the router side, the NI implements a flow control mechanism to avoid network congestion [Sin97]. The FIFOs between the Hub and the network also implement separate virtual FIFOs for requests and replies. The outgoing FIFO has an invalidation destination generator which takes the bit vector of nodes to be invalidated and generates individual message for them, a routing table that pre-determines the routing decisions based on source and destination nodes, and virtual channel selection logic.

8.6.7 Performance Characteristics

The stated hardware bandwidths and occupancies of the system are as follows, assuming a 195MHz processor. Bandwidth on the SysAD bus, shared by two processors in the node, is 780MB/s, as is the bandwidth between the Hub and local memory, and the bandwidth between a node and the network in each direction. The data bandwidth of the local memory itself is about 670 MB/s. The occupancy of the Hub at the home for a transaction on a cache block is about 20 Hub cycles (about 40 processor cycles), though it varies between 18 and 30 Hub cycles depending on whether successive directory pages accessed are in the same bank of the directory SDRAM and on the exact pattern of successive accesses (read or read-exclusives followed by another of the same or followed by a writeback or an uncached operation). The latencies of references depend on many factors such as the type of reference, whether the home is local or not, where and in what state the data are currently cached, and how much contention is there for resources along the way. The latencies can be measured using microbenchmarks. As we did for the Challenge machine in Chapter 6, let us examine microbenchmark results for latency and bandwidth first, followed by the performance and scaling of our six applications.

Characterization with Microbenchmarks

Unlike the MIPS R4400 processor used in the SGI Challenge, the Origin's MIPS R10000 processor is dynamically scheduled and does not stall on a read miss. This makes it more difficult to measure read latency. We cannot for example measure the unloaded latency of a read miss by simply executing the microbenchmark from Chapter 4 that reads the elements of an array with stride greater than the cache block size. Since the misses are to independent locations, subsequent misses will simply be overlapped with one another and will not see their full latency. Instead, this microbenchmark will give us a measure of the throughput that the system can provide on successive read misses issued from a processor. The throughput is the inverse of the "latency" measured by this microbenchmark, which we can call the pipelined latency.

To measure the full latency, we need to ensure that subsequent operations are dependent on each other. To do this, we can use a microbenchmark that chases pointers down a linked list: The address of the next read is not available to the processor until the previous read completes, so the reads cannot be overlapped. However, it turns out this is a little pessimistic in determining the unloaded read latency. The reason is that the processor implements critical word restart; i.e. it can use the value returned by a read as soon as that word is returned to the processor, without waiting for the rest of the cache block to be loaded in the caches. With the pointer chasing microbenchmark, the next read will be issued before the previous block has been loaded, and will contend

with the loading of that block for cache access. The latency obtained from this microbenchmark, which includes this contention, can be called back-to-back latency. Avoiding this contention between successive accesses requires that we put some computation between the read misses that depends on the data being read but that does not access the caches between two misses. The goal is to have this computation overlap the time that it takes for the rest of the cache block to load into the caches after a read miss, so the next read miss will not have to stall on cache access. The time for this overlap computation must of course be subtracted from the elapsed time of the microbenchmark to measure the true unloaded read miss latency assuming critical word restart. We can call this the restart latency [HLK97]. Table 8-1 shows the restart and back-to-back latencies measured on the Origin2000, with only one processor executing the microbenchmark but the data that are accessed being distributed among the memories of different numbers of processors. The back to back latency is usually about 13 bus cycles (133 ns), because the L2 cache block size (128B) is 12 double words longer than the L1 cache block size (32B) and there is one cycle for bus turnaround.

Table 8-1 Back-to-back and restart latencies for different system sizes. The first column shows where in the extended memory hierarchy the misses are satisfied. For the 8P case, for example, the misses are satisfied in the furthest node away from the requestor in a system of 8 processors. Given the Origin2000 topology, with 8 processors this means traversing through two network routers.

Memory level	Network Routers Traversed	Back-to-Back (ns)	Restart (ns)
L1 cache	0	5.5	5.5
L2 cache	0	56.9	56.9
local memory	0	472	329
4P remote memory	1	690	564
8P remote memory	2	890	759
16P remote memory	3	991	862

Table 8-2 lists the back-to-back latencies for different initial states of the block being referenced [HLK97]. Recall that the owner node is home when the block is in unowned or shared state at the directory, and the node that has a cached copy when the block is in exclusive state. The restart latency for the case where both the home and the owner are the local node (i.e. if owned the block is owned by the other processor in the same node) is 338 ns for the unowned state, 656 ns for the clean-exclusive state, and 892 ns for the modified state.

Table 8-2 Back-to-back latencies (in ns) for different initial states of the block. The first column indicates whether the home of the block is local or not, the second indicates whether the current owner is local or not, and the last three columns give the latencies for the block being in different states.

Home	Owner	State of Block		
		Unowned	Clean-Exclusive	Modified
local	local	472	707	1036
remote	local	704	930	1272
local	remote	472	930	1159
remote	remote	704	917	1097

Application Speedups

Figure 8-22 shows the speedups for the six parallel applications on a 32-processor Origin2000,
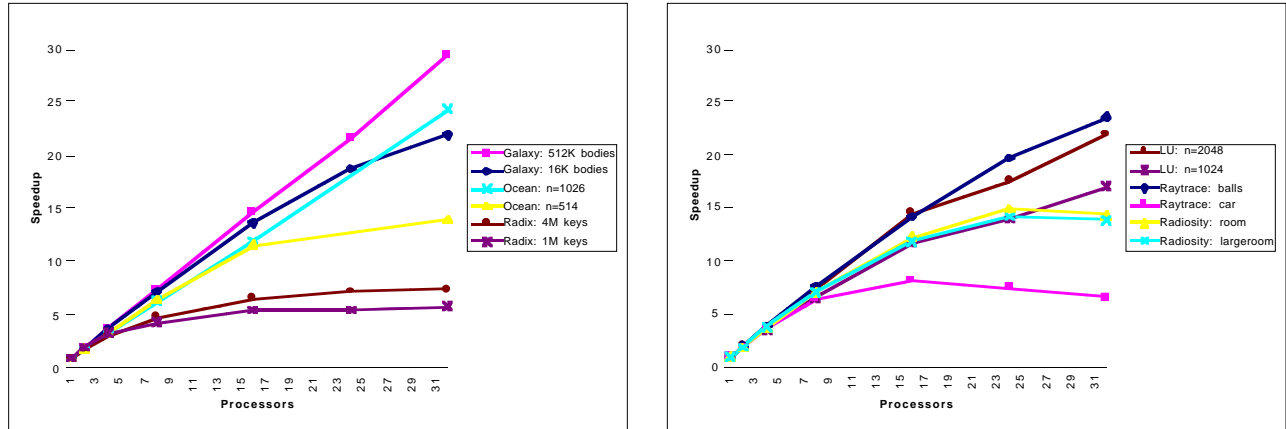
Figure 8-22 Speedups for the parallel applications on the Origin2000.

Two problem sizes are shown for each application. The Radix sorting program does not scale well, and the Radiosity application is limited by the available input problem sizes. The other applications speed up quite well when reasonably large problem sizes are used.

using two problem sizes for each. We see that most of the applications speed up well, especially once the problem size is large enough. The dependence on problem size is particular stark in applications like Ocean and Raytrace. The exceptions to good speedup at this scale are Radiosity and particularly Radix. In the case of Radiosity, it is because even the larger problem is relatively small for a machine of this size and power. We can expect to see better speedups for larger data sets. For Radix, the problem is the highly scattered, bursty pattern of writes in the permutation phase. These writes are mostly to locations that are allocated remotely, and the flood of requests, invalidations, acknowledgments and replies that they generate cause tremendous contention and hot-spotting at Hubs and memories. Running larger problems doesn't alleviate the situation, since the communication to computation ratio is essentially independent of problem size; in fact, it worsens the situation once a processor's partition of the keys do not fit in its cache, at which point writeback transactions are also thrown into the mix. For applications like Radix and an FFT (not shown) that exhibit all-to-all bursty communication, the fact that two processors share a Hub and two Hubs share a router also causes contention at these resources, despite their high bandwidths. For these applications, the machine would perform better if it had only a single processor per Hub and per router. However, the sharing of resources does save cost, and does not get in the way of most of the other applications [JS97].

Scaling

Figure 8-23 shows the scaling behavior for the Barnes-Hut galaxy simulation on the Origin2000. The results are quite similar to those on the SGI Challenge in Chapter 6, although extended to more processors. For applications like Ocean, in which an important working set is proportional to the data set size per processor, machines like the Origin2000 display an interesting effect due to scaling when we start from a problem size where the working set does not fit in the cache on a uniprocessor. Under PC and TC scaling, the data set size per processor diminishes with increasing number of processors. Thus, although the communication to computation ratio increases, we

observe superlinear speedups once the working set starts to fit in the cache, since the performance within each node becomes much better when the working set fits in the cache. Under MC scaling, communication to computation ratio does not change, but neither does the working set size per processor. As a result, although the demands on the communication architecture scale more favorably than under TC or PC scaling, speedups are not so good because the beneficial effect on node performance of the working set suddenly fitting in the cache is no longer observed.
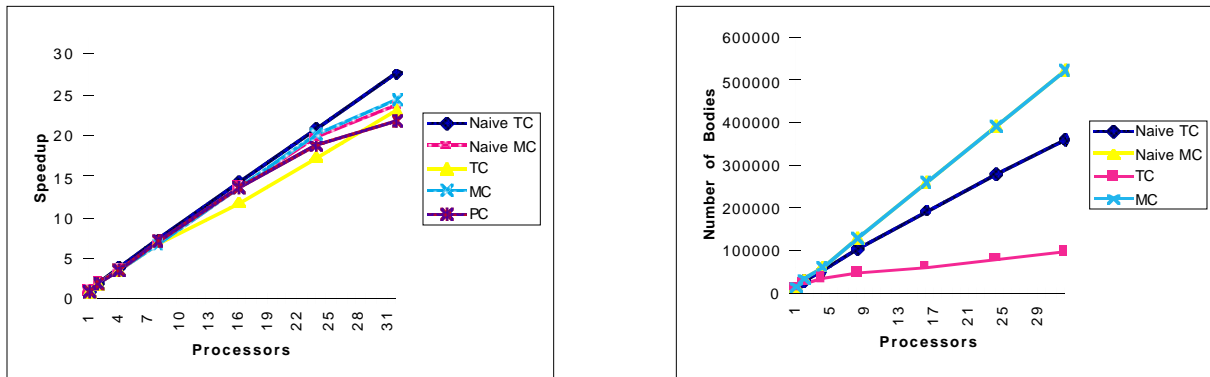
Figure 8-23 Scaling of speedups and number of bodies under different scaling models for Barnes-Hut on the Origin2000.

As with the results for bus-based machines in Chapter 6, the speedups are very good under all scaling models and the number of bodies that can be simulated grows much more slowly under realistic TC scaling than under MC or naive TC scaling.

8.7 Cache-based Directory Protocols: The Sequent NUMA-Q

The flat, cache-based directory protocol we describe is the IEEE Standard Scalable Coherent Interface (SCI) protocol [Gus92]. As a case study of this protocol, we examine the NUMA-Q machine from Sequent Computer Systems, Inc., a machine targeted toward commercial workloads such as databases and transaction processing [CL96]. This machine relies heavily on third-party commodity hardware, using stock SMPs as the processing nodes, stock I/O links, and the Vitesse "data pump" network interface to move data between the node and the network. The only customization is in the board used to implement the SCI directory protocol. A similar directory protocol is also used in the Convex Exemplar series of machines [Con93,TSS+96], which like the SGI Origin are targeted more toward scientific computing.

NUMA-Q is a collection of homogeneous processing nodes interconnected by a high-speed links in a ring interconnection (Figure 8-24). Each processing node is an inexpensive Intel Quad bus-based multiprocessor with four Intel Pentium Pro microprocessors, which illustrates the use of high-volume SMPs as building blocks for larger systems. Systems from Data General [ClA96] and from HAL Computer Systems [WGH+97] also use Pentium Pro Quads as their processing nodes, the former also using an SCI protocol similar to NUMA-Q across Quads and the latter using a memory-based protocol inspired by the Stanford DASH protocol. (In the Convex Exemplar series, the individual nodes connected by the SCI protocol are small directory-based multiprocessors kept coherent by a different directory protocol.) We described the Quad SMP node in Chapter 1 (see Figure 1-17 on page 48) and so shall not discuss it much here.
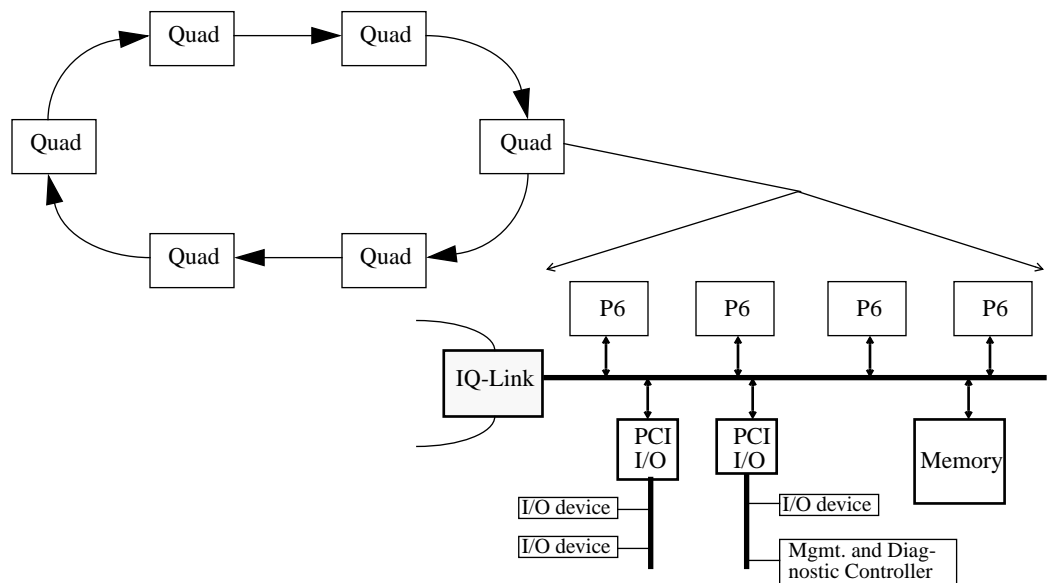
Figure 8-24 Block diagram of the Sequent NUMA-Q multiprocessor.

The IQ-Link board in each Quad plugs into the Quad memory bus, and takes the place of the Hub in the SGI Origin. In addition to the data path from the Quad bus to the network and the directory logic and storage, it also contains an (expandable) 32MB, 4-way set associative remote cache for blocks that are fetched from remote memory. This remote cache is the interface of the Quad to the directory protocol. It is the only cache in the Quad that is visible to the SCI directory proto-col; the individual processor caches are not, and are kept coherent with the remote cache through the snoopy bus protocol within the Quad. Inclusion is also preserved between the remote cache and the processor caches within the node through the snoopy coherence mechanisms, so if a block is replaced from the remote cache it must be invalidated in the processor caches, and if a block is placed in modified state in a processor cache then the state in the remote cache must reflect this. The cache block size of this remote cache is 64 bytes, which is therefore the granular-ity of both communication and coherence across Quads.

8.7.1 Cache Coherence Protocol

Two interacting coherence protocols are used in the Sequent NUMA-Q machine. The protocol used within a Quad to keep the processor caches and the remote cache coherent is a standard bus-based Illinois (MESI). Across Quads, the machine uses the SCI cache-based, linked list directory protocol. Since the directory protocol sees only the per-Quad remote caches, it is for the most part oblivious to how many processors there are within a node and even to the snoopy bus proto-col itself. This section focuses on the SCI directory protocol across remote caches, and ignores the multiprocessor nature of the Quad nodes. Interactions with the snoopy protocol in the Quad will be discussed later.

Directory Structure

The directory structure of SCI is the flat, cache-based distributed doubly-linked list scheme that was described in Section and illustrated in Figure 8-8 on page 527. There is a linked list of sharers per block, and the pointer to the head of this list is stored in the main memory that is the home of the corresponding memory block. An entry in the list corresponds to a remote cache in a Quad. It is stored in Synchronous DRAM memory in the IQ-Link board of that Quad, together with the forward and backward pointers for the list. Figure 8-25 shows a simplified representation of a list. The first element or node is called the head of the list, and the last node the tail. The head node has both read and write permission on its cached block, while the other nodes have only read permission (except in a special-case extension called pairwise sharing that we shall discuss later). The pointer in a node that points to its neighbor in the direction toward the tail of the list is called the forward or downstream pointer, and the other is called the backward or upstream pointer. Let us see how the cross-node SCI coherence protocol uses this directory representation.
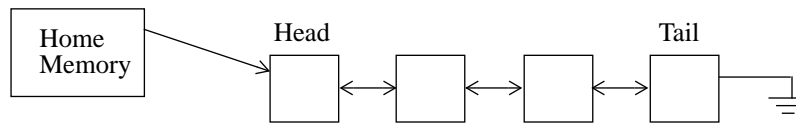
Figure 8-25 An SCI sharing list.

States

A block in main memory can be in one of three directory states whose names are defined by the SCI protocol:

> *Home*: no remote cache (Quad) in the system contains a copy of the block (of course, a processor cache in the home Quad itself may have a copy, since this is not visible to the SCI coherence protocol but is managed by the bus protocol within the Quad).

> *Fresh*: one or more remote caches may have a read-only copy, and the copy in memory is valid.This is like the shared state in Origin.

> *Gone*: another remote cache contains a writable (exclusive-clean or dirty) copy. There is no valid copy on the local node. This is like the exclusive directory state in Origin.

Consider the cache states for blocks in a remote cache. While the processor caches within a Quad use the standard Illinois or MESI protocol, the SCI scheme that governs the remote caches has a large number of possible cache states. In fact, seven bits are used to represent the state of a block in a remote cache. The standard describes twenty nine stable states and many pending or transient states. Each stable state can be thought of as having two parts, which is reflected in the naming structure of the states. The first describes which element of the sharing list for that block that cache entry is. This may be ONLY (for a single-element list), HEAD, TAIL, or MID (neither the head nor the tail of a multiple-element list). The second part describes the actual state of the cached block. This includes states like

> Dirty: modified and writable

> Clean: unmodified (same as memory) but writable

> Fresh: data may be read, but not written until memory is informed

Copy: unmodified and readable

and several others. A full description can be found in the SCI Standard document [IEE93]. We shall encounter some of these states as we go along.

The SCI standard defines three primitive operations that can be performed on a list. Memory operations such as read misses, write misses, and cache replacements or writebacks are implemented through these primitive operations, which will be described further in our discussion:

List construction: adding a new node (sharer) to the head of a list,

Rollout: removing a node from a sharing list, which requires that a node communicate with its upstream and downstream neighbors informing them of their new neighbors so they can update their pointers, and

Purging (invalidation): the node at the head may purge or invalidate all other nodes, thus resulting in a single-element list. Only the head node can issue a purge.

The SCI standard also describes three "levels" of increasingly sophisticated protocols. There is a *minimal* protocol that does not permit even read-sharing; that is, only one node at a time can have a cached copy of a block. Then there is a *typical* protocol which is what most systems are expected to implement. This has provisions for read sharing (multiple copies), efficient access to data that are in FRESH state in memory, as well as options for efficient DMA transfers and robust recovery from errors. Finally, there is the *full* protocol which implements all of the options defined by the standard, including optimizations for pairwise sharing between only two nodes and queue-on-lock-bit (QOLB) synchronization (see later). The NUMA-Q system implements the typical set, and this is the one we shall discuss. Let us see how these states and the three primitives are used to handle different types of memory operations: read requests, write requests, and replacements (including writebacks). In each case, a request is first sent to the home node for the block, whose identity is determined from the address of the block. We will initially ignore the fact that there are multiple processors per Quad, and examine the interactions with the Quad protocol later.

Handling Read Requests

Suppose the read request needs to be propagated off-Quad. We can now think of this node's remote cache as the requesting cache as far as the SCI protocol is concerned. The requesting cache first allocates an entry for the block if necessary, and sets the state of the block to a pending (busy) state; in this state, it will not respond to other requests for that block that come to it. It then begins a list-construction operation to add itself to the sharing list, by sending a request to the home node. When the home receives the request, its block may be in one of three states: HOME, FRESH and GONE.

HOME: There are no cached copies and the copy in memory is valid. On receiving this request, the home updates its state for the block to FRESH, and set its head pointer to point to the requesting node. The home then replies to the requestor with the data, which upon receipt updates its state from PENDING to ONLY_FRESH. All actions at nodes in response to a given transaction are atomic (the processing for one is completed before the next one is handled), and a strict request-reply protocol is followed in all cases.

FRESH: There is a sharing list, but the copy at the home is valid. The home changes its head pointer to point to the requesting cache instead of the previous head of the list. It then sends back a transaction to the requestor containing the data and also a pointer to the previous head.

On receipt, the requestor moves to a different pending state and sends a transaction to that previous head asking to be attached as the new head of the list. The previous head reacts to

(a) Initial state, plus first round-trip

(a) After first round-trip

(a) Second round-trip

(a) Final state
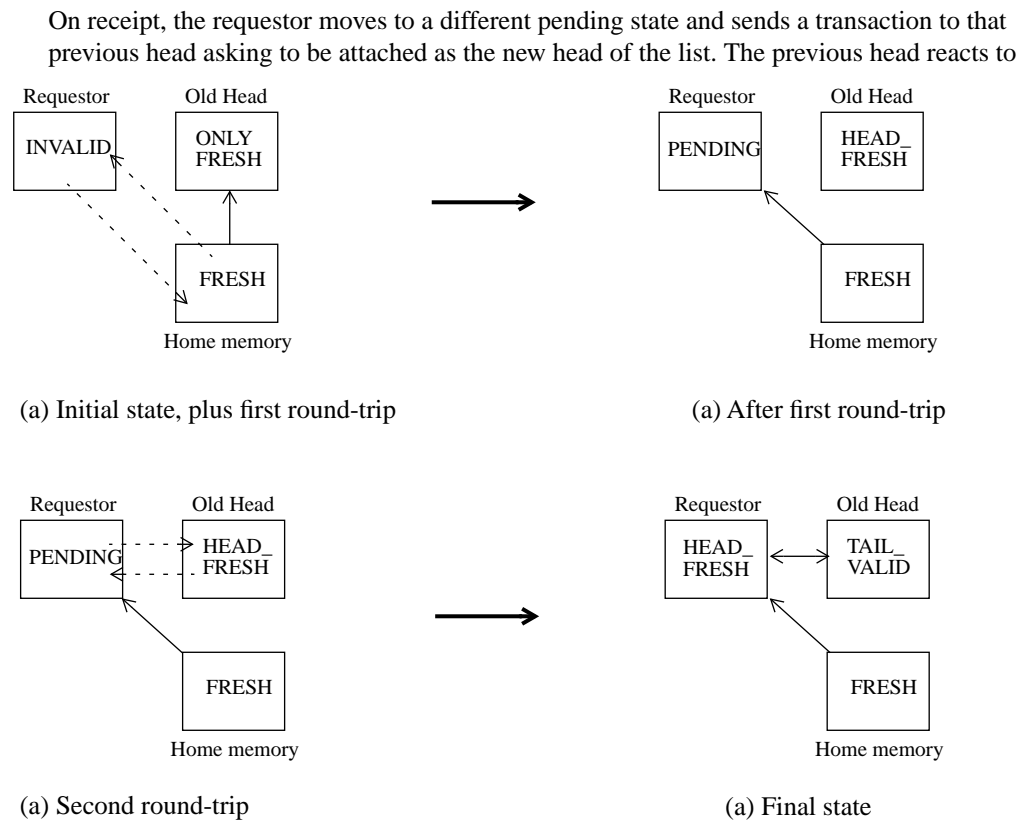
Figure 8-26 Messages and state transitions in a read miss to a block that is in the FRESH state at home, with one node on the sharing list.

Solid lines are the pointers in the sharing list, while dotted lines represent network transactions. Null pointers are not shown.

this message by changing its state from HEAD_FRESH to MID_VALID or from ONLY_FRESH to TAIL_VALID as the case may be, updating its backward pointer to point to the requestor, and sending an acknowledgment to the requestor. When the requestor receives this acknowledgment it sets its forward pointer to point to the previous head and changes its state from the pending state to HEAD_FRESH. The sequence of transactions and actions is shown in Figure 8-26, for the case where the previous head is in state HEAD_FRESH when the request comes to it.

GONE: The cache at the head of the sharing list has an exclusive (clean or modified) copy of the block. Now, the memory does not reply with the data, but simply stays in the GONE state and sends a pointer to the previous head back to the requestor. The requestor goes to a new pending state and sends a request to the previous head asking for the data and to attach to the head of the list. The previous head changes its state from HEAD_DIRTY to MID_VALID or ONLY_DIRTY to TAIL_VALID or whatever is appropriate, sets its backward pointer to point to the requestor and returns the data to the requestor. (the data may have to be retrieved from one of the processor caches in the previous head node). The requestor then updates its copy, sets its state to HEAD_DIRTY, and sets its forward pointer to point to the new head, as always in a single atomic action. Note that even though the reference was a read, the head of

the sharing list is in HEAD_DIRTY state. This does not have the standard meaning of dirty that we are familiar with; that is, that it can write those data without having to invalidate anybody. It means that it can indeed write the data without communicating with the home (and even before sending out the invalidations), but it must invalidate the other nodes in the sharing list since they are in valid state.

It is possible to fetch a block in HEAD_DIRTY state even when the directory state is not GONE, for example when that node is expected to write that block soon afterward. In this case, if the directory state is FRESH the memory returns the data to the requestor, together with a pointer to the old head of the sharing list, and then puts itself in GONE state. The requestor then prepends itself to the sharing list by sending a request to the old head, and puts itself in the HEAD_DIRTY state. The old head changes its state from HEAD_FRESH to MID_VALID or from ONLY_FRESH to TAIL_VALID as appropriate, and other nodes on the sharing list remain unchanged.

In the above cases where a requestor is directed to the old head, it is possible that the old head (let's call it A) is in pending state when the request from the new requestor (B) reaches it, since it may have some transaction outstanding on that block. This is dealt with by extending the sharing list backward into a (still distributed) *pending list*. That is, node B will indeed be physically attached to the head of the list, but in a pending state waiting to truly become the head. If another node C now makes are request to the home, it will be forwarded to node B and will also attach itself to the pending list (the home will now point to C, so subsequent requests will be directed there). At any time, we call the "true head" (here A) the head of the sharing list, the part of the list before the true head the pending list, and the latest element to have joined the pending list (here C) the pending head (see Figure 8-27). When A leaves the pending state and completes its opera-
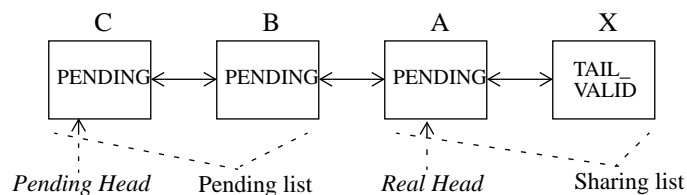
Figure 8-27 Pending lists in the SCI protocol.

tion, it will pass on the "true head" status to B, which will in turn pass it on to C when its request is completed. The alternative to using a pending list would have been to NACK requests that come to a cache that is in pending state and have the requestor retry until the previous head's pending state changes. We shall return to these issues and choices when we discuss dealing with correctness issues in Section 8.7.2. Note also that there is no pending or busy state at the directory, which always simply takes atomic actions to change its state and head pointer and returns the previous state/pointer information to the requestor, a point we will revisit there.

Handling Write Requests

The head node of a sharing list is assumed to always have the latest copy of the block (unless it is in a pending state). Thus, only the head node is allowed to write a block and issues invalidations. On a *write* miss, there are three cases. The first case is when the writer is already at the head of

the list, but it does not have the sole modified copy. In this case, it first ensures that it is in the appropriate state for this case, by communicating with the home if necessary (and in the process ensuring that the home block is in or transitions to the GONE state). It then modifies the data locally, and invalidates the rest of the nodes in the sharing list. This case will be elaborated on in the next two paragraphs. The second case is when the writer is not in the sharing list at all. In this case, the writer must first allocate space for and obtain a copy of the block, then add itself to the head of the list using the list construction operation, and then perform the above steps to complete the write. The third case is when the writer is in the sharing list but not at the head. In this case, it must remove itself from the list (rollout), then add itself to the head (list construction), and finally perform the above steps. We shall discuss rollout in the context of replacement, where it is also needed, and we have already seen list construction. Let us focus on the case where the writing node is already at the head of the list.

If the block is in HEAD_DIRTY state in the writer's cache, it is modified right away (since the directory must be in GONE state) and then the writing node purges the rest of the sharing list. This is done in a serialized request-reply manner: An invalidation request is sent to the next node in the sharing list, which rolls itself out from the list and sends back to the head a pointer to the next node in the list. The head then sends this node a similar request, and so on until all entries are purged (i.e. the reply to the head contains a null pointer, see also Figure 8-28). The writer, or
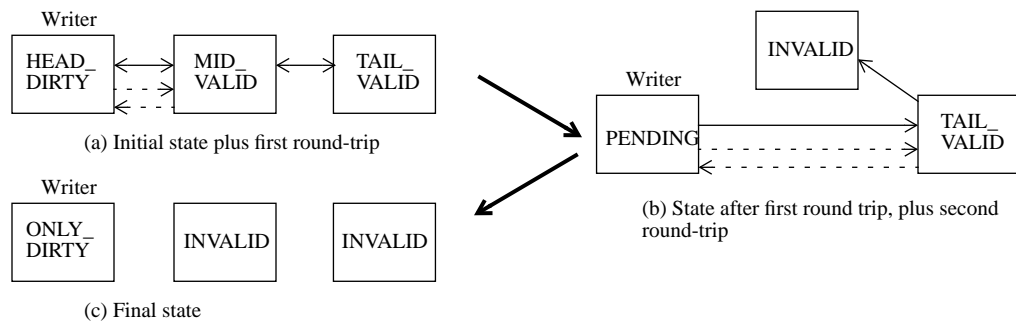
Figure 8-28 Purging a sharing list from a HEAD_DIRTY node in SCI.

head node, stays in a special pending state while the purging is in progress. During this time, new attempts to add to the sharing list are delayed in a pending list as usual. The latency of purging a sharing list is a few round trips (invalidation request, acknowledgment and the rollout transactions) plus the associated actions *per sharing list entry*, so it is important that sharing lists not get too long. It is possible to reduce the number of network transactions in the critical path by having each node pass on an invalidation request to the next node rather than return the identity to the writer. This is not part of the SCI standard since it complicates protocol-level recovery from errors by distributing the state of the invalidation progress, but practical systems may be tempted to take advantage of this short-cut.

If the writer is the head of the sharing list but has the block in HEAD_FRESH state, then it must be changed to HEAD_DIRTY before the block can be modified and the rest of the entries purged. The writer goes into a pending state and sends a request to the home, the home changes from FRESH to GONE state and replies to the message, and then the writer goes into a different pending state and purges the rest of the blocks as above. It may be that when the request reaches the home the home is no longer in FRESH state, but it points to a newly queued node that got there in

the meantime and has been directed to the writer. When the home looks up its state, it detects this situation and sends the writer a corresponding reply that is like a NACK. When the writer receives this reply, based on its local pending state it deletes itself from the sharing list (how it does this given that a request is coming at it is discussed in the next subsection) and tries to reattach as the head in HEAD_DIRTY or ONLY_DIRTY state by sending the appropriate new request to the home. Thus, it is not a retry in the sense that it does not try the same request again, but a suitably modified request to reflect the new state of itself and the home. The last case for a write is if the writer has the block in ONLY_DIRTY state, in which case it can modify the block without generating any network transactions.

Handling Writeback and Replacement Requests

A node that is in a sharing list for a block may need to delete itself, either because it must become the head in order to perform a write operation, or because it must be replaced to make room for another block in the cache for capacity or conflict reasons, or because it is being invalidated. Even if the block is in shared state and does not have to write data back, the space in the cache (and the pointer) will now be used for another block and its list pointers, so to preserve a correct representation the block being replaced must be removed from its sharing list. These replacements and list removals use the rollout operation.

Consider the general case of a node trying to roll out from the middle of a sharing list. The node first sets itself to a special pending state, then sends a request each to its upstream and downstream neighbors asking them to update their forward and backward pointers, respectively, to skip that node. The pending state is needed since there is nothing to prevent two adjacent nodes in a sharing list from trying to roll themselves out at the same time, which can lead to a race condition in the updating of pointers. Even with the pending state, if two adjacent nodes indeed try to roll out at the same time they may set themselves to pending state simultaneously and send messages to each other. This can cause deadlock, since neither will respond while they are in locked state. A simple priority system is used to avoid such deadlock: By convention, the node closer to the tail of the list has priority and is deleted first. The neighbors do not have to change their state, and the operation is completed by setting the state of the replaced cache entry to invalid when both the neighbors have replied. If the element being replaced is the last element of a two-element list, then the head of the list may change its state from HEAD_DIRTY or HEAD_FRESH to ONLY_DIRTY or ONLY_FRESH as appropriate.

If the entry to be rolled out is the head of the list, then the entry may be in dirty state (a writeback) or in fresh state (a replacement). Here too, the same set of transactions is used whether it is a writeback or merely a replacement. The head puts itself in a special rollout pending state and first sends a transaction to its downstream neighbor. This causes the latter to set its backward pointer to the home memory and change its state appropriately (e.g. from TAIL_VALID or MID_VALID to HEAD_DIRTY, or from MID_FRESH to HEAD_FRESH). When the replacing node receives a reply, it sends a transaction to the home which updates its pointer to point to the new head but need not change its state. The home replies to the replacer, which is now out of the list and sets its state to INVALID. Of course, if the head is the only node in the list then it needs to communicate only with memory, which will set its state to HOME.

In a distributed protocol like SCI, it is possible that a node sends a request to another node, but when it arrives the local state at the recipient is not compatible with the incoming request (another transaction may have changed the state in the interim). We saw one example in the pre-

vious subsection, with a write to a block in HEAD_FRESH state in the cache, and the above situation provides another. For one thing, by the time the message from the head that is trying to replace itself (let's call it the replacer) gets to the home, the home may have set its head pointer to point to a different node X, from which it has received a request for the block in the interim. In general, whenever a transaction comes in, the recipient looks up its local state and the incoming request type; if it detects a mismatch, it does not perform the operation that the request solicits but issues a reply that is a lot like a NACK (plus perhaps some other information). The requestor will then check its local state again and take an appropriate action. In this specific case, the home detects that the incoming transaction type requires that the requestor be the current head, which is not true so it NACKs the request. The replacer keeps retrying the request to the home, and being NACKed. At some point, the request from X that was pointed to the replacer will reach the replacer asking to be prepended to the list. The replacer will look up its (pending) state, and reply to that request telling it to instead go to the downstream neighbor (the real head, since the replacer is rolling out of the list). The replacer is now off the list and in a different pending state, waiting to go to INVALID state, which it will when the next NACK from the home reaches it. Thus, the protocol does include NACKs, though not in the traditional sense of asking requests to retry when a node or resource is busy. NACKs are just to indicate inappropriate requests and facilitate changes of state at the requestor; a request that is NACKed will never succeed in its original form, but may cause a new type of request to be generated which may succeed.

Finally, a performance question when a block needs to be written back upon a miss is whether the miss should be satisfied first or the block should be written back first. In discussing bus-based protocols, we saw that most often the miss is serviced first, and the block to be written back is put in a writeback buffer (which must also be snooped). In NUMA-Q, the simplifying decision is made to service the writeback (rollout) first, and only then satisfy the miss. While this slows down misses that cause writebacks, the complexity of the buffering solution is greater here than in bus-based systems. More importantly, the replacements we are concerned with here are from the remote cache, which is large enough (tens of megabytes) that replacements are likely to be very infrequent.

8.7.2 Dealing with Correctness Issues

A major emphasis in the SCI standard is providing well-defined, uniform mechanisms for preserving serialization, resolving race conditions, and avoiding deadlock, livelock and starvation. The standard takes a stronger position on these issues, particularly starvation and fairness, than many other coherence protocols. We have mentioned earlier that most of the correctness considerations are satisfied by the use of distributed lists of sharers as well as pending requests, but let us look at how this works in one further level of detail.

Serialization of Operations to a Given Location

In the SCI protocol too, the home node is the entity that determines the order in which cache misses to a block are serialized. However, here the order is that in which the requests *first arrive* at the home, and the mechanism used for ensuring this order is very different. There is no busy state at the home. Generally, the home accepts every request that comes to it, either satisfying it wholly by itself or directing it to the node that it sees as the current head of the sharing list (this may in fact be the true head of the sharing list or just the head of the pending list for the line). Before it directs the request to another node, it first updates its head pointer to point to the current requestor. The next request for the block from any node will see the updated state and pointer—

i.e. to the previous requestor—even though the operation corresponding to the previous request is not globally complete. This ensures that the home does not direct two conflicting requests for a block to the same node at the same time, avoiding race conditions. As we have seen, if a request cannot be satisfied at the previous head node to which it was directed—i.e. if that node is in pending state—the requestor will attach itself to the distributed pending list for that block and await its turn as long as necessary (see Figure 8-27 on page 571). Nodes in the pending list obtain access to the block in FIFO order, ensuring that the order in which they complete is the same as that in which they first reached the home.

In some cases, the home may NACK requests. However, unlike in the earlier protocols, we have seen that those requests will never succeed in their current form, so they do not count in the serialization. They may be modified to new, different requests that will succeed, and in that case those new requests will be serialized in the order in which they first reach the home.

Memory Consistency Model

While the SCI standard defines a coherence protocol and a transport layer, including a network interface design, it does not specify many other aspects like details of the physical implementation. One of the aspects that it does not prescribe is the memory consistency model. Such matters are left to the system implementor. NUMA-Q does not satisfy sequential consistency, but rather more relaxed memory consistency model called processor consistency that we shall discuss in Section 9.2, since this is the consistency model supported by the underlying microprocessor.

Deadlock, Livelock and Starvation

The fact that a distributed pending list is used to hold waiting requests at the requestors themselves, rather than a hardware queue shared at the home node by all blocks allocated in it, implies that there is no danger of input buffers filling up and hence no deadlock problem at the protocol level. Since requests are not NACKed from the home (unless they are inappropriate and must be altered) but will simply join the pending list and always make progress, livelock does not occur. The list mechanism also ensures that the requests are handled in FIFO order as they first come to the home, thus preventing starvation.

The total number of pending lists that a node can be a part of is the number of requests it can have outstanding, and the storage for the pending lists is already available in the cache entries (replacement of a pending entry is not allowed, the processor stalls on the access that is causing this replacement until the entry is no longer pending). Queuing and buffering issues at the lower, transport level are left up to the implementation; the SCI standard does not take a position on them, though most implementations including NUMA-Q use separate request and reply queues on each of the incoming and outgoing paths.

Error Handling

The SCI standard provides some options in the "typical" protocol to recover from errors at the protocol level. NUMA-Q does not implement these, but rather assumes that the hardware links are reliable. Standard ECC and CRC checks are provided to detect and recover from hardware errors in the memory and network links. Robustness to errors at the protocol level often comes at the cost of performance. For example, we mentioned earlier that SCI's decision to have the writer send all the invalidations one by one serialized by replies simplifies error recovery (the writer

knows how many invalidations have been completed when an error occurs) but compromises performance. NUMA-Q retains this feature, but it and other systems may not in future releases.

8.7.3 Protocol Extensions

While the SCI protocol is fair and quite robust to errors, many types of operations can generate many serialized network transactions and therefore become quite expensive. A read miss requires two network transactions with the home, and at least two with the head node if there is one, and perhaps more with the head node if it is in pending state. A replacement requires a rollout, which again is four network transactions. But the most troublesome operation from a scalability viewpoint is invalidation on a write. As we have seen, the cost of the invalidation scales linearly with the number of nodes on the sharing list, with a fairly large constant (more than a round-trip time), which means that writes to widely shared data do not scale well at all. Other performance bottlenecks arise when many requests arrive for the same block and must be added to a pending list; in general, the latency of misses tends to be larger in SCI than in memory-based protocols. Extensions have been proposed to SCI to deal with widely shared data, building a combining hierarchy through the bridges or switches that connect rings. Some extensions require changes to the basic protocol and hardware structures, while others are plug-in compatible with the basic SCI protocol and only require new implementations of the bridges. The complexity of the extensions tends to reduce performance for low degrees of sharing. They are not finalized in the standard, are quite complex, and are beyond the scope of this discussion. More information can be found in [IEE95, KaG96, Kax96]. One extension that is included in the standard specializes the protocol for the case where only two nodes share a cache block and they ping-pong ownership of it back and forth between them by both writing it repeatedly. This is described in the SCI protocol document [IEE93].

Unlike Origin, NUMA-Q does not provide hardware or OS support for automatic page migration. With the very large remote caches, capacity misses needing to go off-node are less of an issue. However, proper page placement can still be useful when a processor writes and has to obtain ownership for data: If nobody else has a copy (for example the interior portion of a processor's partition in the equation solver kernel or in Ocean) then if the page is allocated locally obtaining ownership does not generate network traffic while if it is remote it does. The NUMA-Q position is that data migration in main memory is the responsibility of user-level software. Similarly, little hardware support is provided for synchronization beyond simple atomic exchange primitives like test&set.

An interesting aspect of the NUMA-Q in terms of coherence protocols is that it has two of them: a bus-based protocol within the Quad and an SCI directory protocol across Quads, isolated from the Quad protocol by the remote cache. The interactions between these two protocols are best understood after we look more closely at the organization of the IQ-Link.

8.7.4 Overview of NUMA-Q Hardware

Within a Quad, the second-level caches currently shipped in NUMA-Q systems is 512KB large and 4-way set associative with a 32-byte block size. The Quad bus is a a 532 MB/s split-transaction in-order bus, with limited facilities for the out-of-order responses needed by a two-level coherence scheme. A Quad also contains up to 4GB of globally addressible main memory, two 32-bit wide 133MB/s Peripheral Component Interface (PCI) buses connected to the Quad bus by PCI bridges and to which I/O devices and a memory and diagnostic controller can attach, and the

IQ-Link board that plugs into the memory bus and includes the communication assist and the network interface.

In addition to the directory information for locally allocated data and the tags for remotely allocated but locally cached data, which it keeps on both the bus side and the directory side, the IQ-Link board consists of four major functional blocks as shown in Figure 8-29: the bus interface controller, the SCI link interface controller, the DataPump, and the RAM arrays. We introduce these briefly here, though they will be described further when discussing the implementation of the IQ-Link in Section 8.7.6.
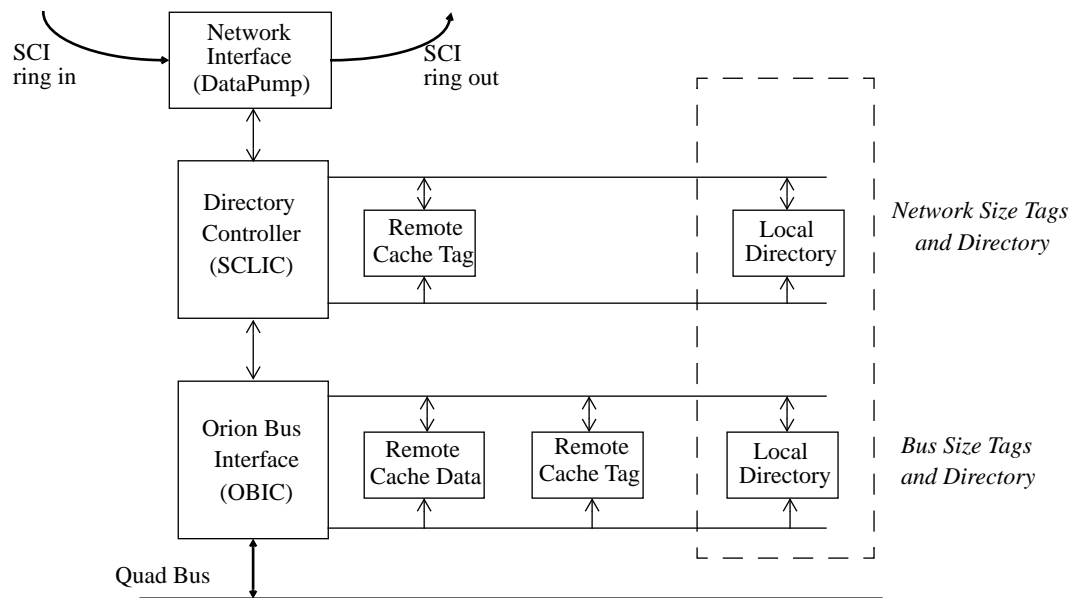
Figure 8-29 Functional block diagram of the NUMA-Q IQ-Link board.

The remote cache data is implemented in Synchronous DRAM (SDRAM). The bus-side tags and directory are implemented in Static RAM (SRAM), while the network side tags and directory can afford to be slower and are therefore implemented in SDRAM.

The *Orion bus interface controller (OBIC)* provides the interface to the shared Quad bus, managing the remote cache data arrays and the bus snooping and requesting logic. It acts as both a pseudo-memory controller that translates accesses to nonlocal data as well as a pseudo-processor that puts incoming transactions from the network on to the bus.

The *DataPump*, a gallium arsenide chip built by Vitesse Semiconductor Corporation, provides the link and packet level transport protocol of the SCI standard. It provides an interface to a ring interconnect, pulling off packets that are destined for its Quad and letting other packets go by.

The *SCI link interface controller (SCLIC)* interfaces to the DataPump and the OBIC, as well as to the interrupt controller and the directory tags. It's main function is to manage the SCI coherence protocol, using one or more programmable protocol engines.

For the interconnection across Quads, the SCI standard defines both a transport layer and a cache coherence protocol. The transport layer defines a functional specification for a node-to-network interface, and a network topology that consists of rings made out of point-to-point links. In particular, it defines a 1GB/sec ring interconnect, and the transactions that can be generated on it. The NUMA-Q system is initially a single-ring topology of up to eight Quads as shown in Figure 8-24, and we shall assume this topology in discussing the protocol (though it does not really matter for the basic protocol). Cables from the quads connect to the ports of a ring, which is contained in a single box called the *Lash*. Larger systems will include multiple eight-quad systems connected together with local area networks. In general, the SCI standard envisions that due to the high latency of rings larger systems will be built out of multiple rings interconnected by switches. The transport layer itself will be discussed in Chapter 10.

Particularly since the machine is targeted toward database and transaction processing workloads, I/O is an important focus of the design. As in Origin, I/O is globally addressable, so any processor can write to or read from any I/O device, not just those attached to the local Quad. This is very convenient for commercial applications, which are often not structured so that a processor need only access its local disks. I/O devices are connected to the two PCI busses that attach through PCI bridges to the Quad bus. Each PCI bus is clocked at half the speed of the memory bus and is half as wide, yielding roughly one quarter the bandwidth. One way for a processor to access I/O devices on other Quads is through the SCI rings, whether through the cache coherence protocol or through uncached writes, just as Origin does through its Hubs and network. However, bandwidth is a precious resource on a ring network. Since commercial applications are often not structured to exploit locality in their I/O operations, I/O transfers can occupy substantial bandwidth, interfering with memory accesses. NUMA-Q therefore provides a separate communication substrate through the PCI busses for inter-quad IO transfers. A "Fiber Channel" link connects to a PCI bus on each node. The links are connected to the shared disks in the system through either point to point connections, an arbitrated Fibre Channel loop, or a Fibre Channel Switch, depending on the scale of the processing and I/O systems (Figure 8-30).

Fibre Channel talks to the disks through a bridge that converts Fibre Channel data format to the SCSI format that the disks accept at 60MB/s sustained. I/O to any disk in the system usually takes a path through the local PCI bus and the Fibre Channel switch; however, if this path fails for some reason the operating system can cause I/O transfers to go through the SCI ring to another Quad, and through its PCI bus and Fibre Channel link to the disk. Processors perform I/O through reads and writes to PCI devices; there is no DMA support in the system. Fibre Channel may also be used to connect multiple NUMA-Q systems in a loosely coupled fashion, and to have multiple systems share disks. Finally, a management and diagnostic controller connects to a PCI bus on each Quad; these controllers are linked together and to a system console through a private local area network like Ethernet for system maintenance and diagnosis.

8.7.5 Protocol Interactions with SMP Node

The earlier discussion of the SCI protocol ignored the multiprocessor nature of the Quad node and the bus-based protocol within it that keeps the processor caches and the remote cache coherent. Now that we understand the hardware structure of the node and the IQ-Link, let us examine the interactions of the two protocols, the requirements that the interacting protocols place upon the Quad, and some particular problems raised by the use of an off-the-shelf SMP as a node.
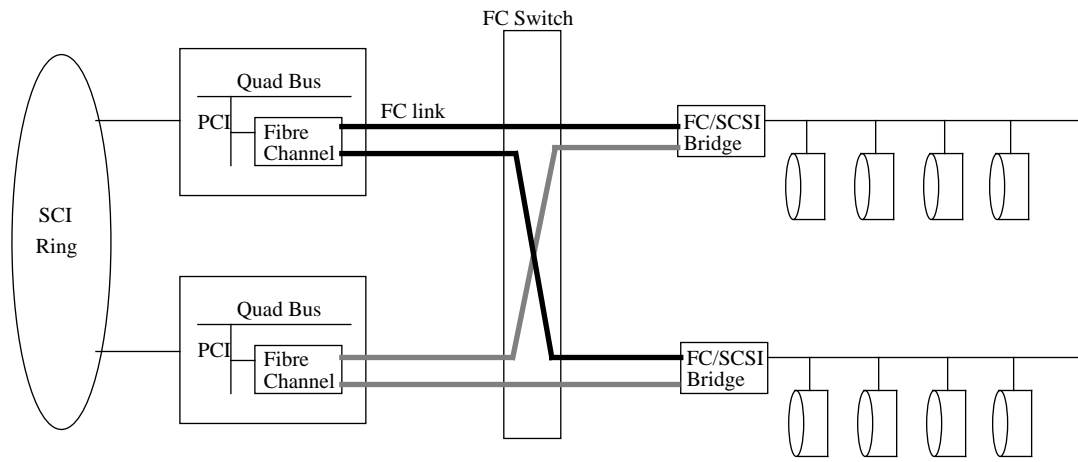
Figure 8-30 I/O Subsystem of the Sequent NUMA-Q.

A read request illustrates some of the interactions. A read miss in a processor's second-level cache first appears on the Quad bus. In addition to being snooped by the other processor caches, it is also snooped by the OBIC. The OBIC looks up the remote cache as well as the directory state bits for locally allocated blocks, to see if the read can be satisfied within the Quad or must be propagated off-node. In the former case, main memory or one of the other caches satisfies the read and the appropriate MESI state changes occur (snoop results are reported after a fixed number, four, of bus cycles; if a controller cannot finish its snoop within this time, it asserts a stall signal for another two bus cycles after which memory checks for the snoop result again; this continues until all snoop results are available). The Quad bus implements in-order responses to requests. However, if the OBIC detects that the request must be propagated off-node, then it must intervene. It does this by asserting a *delayed reply* signal, telling the bus to violate it's in-order response property and proceed with other transactions, and that it will take responsibility for replying to this request. This would not have been necessary if the Quad bus implemented out-of-order replies. The OBIC then passes on the request to the SCLIC to engage the directory protocol. When the reply comes back it will be passed from the SCLIC back to the OBIC, which will place it on the bus and complete the deferred transaction. Note that when extending any bus-based system to be the node of a larger cache-coherent machine, it is essential that the bus be split-transaction. Otherwise the bus will be held up for the entire duration of a remote transaction, not allowing even local misses to complete and not allowing incoming network transactions to be serviced (potentially causing deadlock).

Writes take a similar path out of and back into a Quad. The state of the block in the remote cache, snooped by the OBIC, indicates whether the block is owned by the local Quad or must be propagated to the home through the SCLIC. Putting the node at the head of the sharing list and invalidating other nodes, if necessary, is taken care of by the SCLIC. When the SCLIC is done, it places a response on the Quad bus which completes the operation. An interesting situation arises due to a limitation of the Quad itself. Consider a read or write miss to a locally allocated block that is cached remotely in a modified state. When the reply returns and is placed on the bus as a deferred reply, it should update the main memory. However, the Quad memory was not imple-

mented for deferred requests and replies, and does not update itself on seeing a deferred reply. When a deferred reply is passed down through the OBIC, it must also ensure that it updates the memory through a special action before it gives up the bus. Another limitation of the Quad is its bus protocol. If two processors in a Quad issue read-exclusive requests back to back, and the first one propagates to the SCLIC, we would like to be able to have the second one be buffered and accept the reply from the first in the appropriate state. However, the Quad bus will NACK the second request, which will then have to retry until the first one returns and it succeeds.

Finally, consider serialization. Since serialization at the SCI protocol level is done at the home, incoming transactions at the home have to be serialized not only with respect to one another but also with respect to accesses by the processors in the home Quad. For example, suppose a block is in the HOME state at the home. At the SCI protocol level, this means that no remote cache in the system (which must be on some other node) has a valid copy of the block. However, this does not mean that no processor cache in the home node has a copy of the block. In fact, the directory will be in HOME state even if one of the processor caches has a dirty copy of the block. A request coming in for a locally allocated block must therefore be broadcast on the Quad bus as well, and cannot be handled entirely by the SCLIC and OBIC. Similarly, an incoming request that makes the directory state change from HOME or FRESH to GONE must be put on the Quad bus so that the copies in the processor caches can be invalidated. Since both incoming requests and local misses to data at the home all appear on the Quad bus, it is natural to let this bus be the actual serializing agent at the home.

Similarly, there are serialization issues to be addressed in a requesting quad for accesses to remotely allocated blocks. Activities within a quad relating to remotely allocated blocks are serialized at the local SCLIC. Thus, if there are requests from local processors for a cached block and also requests from the SCI interconnect for the same block coming in to the local node, these requests are serialized at the local SCLIC. Other interactions with the node protocol will be discussed when we have better understood the implementation of the IQ-Link board components.

8.7.6 IQ-Link Implementation

Unlike the single-chip Hub in Origin, the CMOS SCLIC directory controller, the CMOS OBIC bus interface controller and the GaAs Data Pump are separate chips. The IQ-Link board also contains some SRAM and SDRAM chips for tags, state and remote cache data (see Figure 8-29 on page 577). In contrast, the HAL S1 system [WGH+97], discussed later, integrates the entire coherence machinery on a single chip, which plugs right in to the Quad bus. The remote cache data are directly accessible by the OBIC. Two sets of tags are used to reduce communication between the SCLIC and the OBIC, the network-side tags for access by the SCLIC and the bus-side tags for access by the OBIC. The same is true for the directory state for locally allocated blocks. The bus-side tags and directory contain only the information that is needed for the bus snooping, and are implemented in SRAM so they can be looked up at bus speed. The network-side tags and state need more information and can be slower, so they are implemented in synchronous DRAM (SDRAM). Thus, the bus-side local directory SRAM contains only the two bits of directory state per 64-byte block (HOME, FRESH and GONE), while the network side directory contains the 6-bit SCI head pointer as well. The bus side remote cache tags have only four bits of state, and do not contain the SCI forward and backward list pointers. They actually keep track of 14 states, some of which are transient states that ensure forward progress within the quad (e.g. keep track of blocks that are being rolled out, or keep track of the particular bus agent that has an outstanding retry and so must get priority for that block). The network-side tags, which are part

of the directory protocol, do all this, so they contain 7 bits of state plus two 6-bit pointers per block (as well as the 13-bit cache tags themselves).

Unlike the hard-wired protocol tables in Origin, the SCLIC coherence controller in NUMA-Q is programmable. This means the protocol can be written in software or firmware rather than hard-wired into a finite state machine. Every protocol-invoking operation from a processor, as well as every incoming transaction from the network invokes a software "handler" or task that runs on the protocol engine. These software handlers, written in microcode, may manipulate directory state, put interventions on the Quad bus, generate network transactions, etc. The SCLIC has multiple register sets to support 12 read/write/invalidate transactions and one interrupt transaction concurrently (the SCLIC provides a bridge for routing standard Quad interrupts between Quads; some available extra bits are used to include the destination Quad number when generating an interrupt, so the SCLIC can use the standard Quad interrupt interface). A programmable engine is chosen so that the protocol may be debugged in software and corrected by simply downloading new protocol code, so that there would be flexibility to experiment with or change protocols even after the machine was built and bottlenecks discovered, and so that code can be inserted into the handlers to monitor chosen events for performance debugging. Of course, a programmable protocol engine has higher occupancy per transaction than a hard-wired one, so there is a performance cost associated with this decision. Attempts are therefore made to reduce this performance impact: The protocol processor has a simple three-stage pipeline, and issues up to two instructions (a branch and another instruction) every cycle. It uses a cache to hold recently used directory state and tag information rather than access the directory RAMs every time, and is specialized to support the kinds of the bit-field manipulation operations that are commonly needed in directory protocols as well as useful instructions to speed up handler dispatch and management like "queue on buffer full" and "branch on queue space available". A somewhat different programmable protocol engine is used in the Stanford FLASH multiprocessor [KOH+94].

Each Pentium Pro processor can have up to four requests outstanding. The Quad bus can have
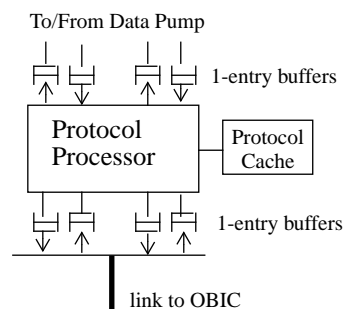
Figure 8-31 Simplified block diagram of SCLIC chip.

eight requests outstanding at a time, like the SGI Challenge bus, and replies come in order. The OBIC can have four requests outstanding (to the SCLIC), and can buffer two incoming transactions to the Quad bus at a time. Thus, if more than four requests on the Quad bus need to go off-Quad, the OBIC will cause the bus to stall. This scenario is not very likely, though. The SCLIC can have up to eight requests outstanding and can buffer four incoming requests at a time. A simplified picture of the SCLIC is shown in Figure 8-31. Finally, the Data Pump request and reply buffers are each two entries deep outgoing to the network and four entries deep incoming. All

request and reply buffers, whether incoming or outgoing, are physically separate in this implementation.

All three components of the IQ-Link board also provide performance counters to enable nonintrusive measurement of various events and statistics. There are three 40-bit memory-mapped counters in the SCLIC and four in the OBIC. Each can be set in software to count any of a large number of events, such as protocol engine utilization, memory and bus utilization, queue occupancies, the occurrence of SCI command types, and the occurrence of transaction types on the P6 bus. The counters can be read by software at any time, or can be programmed to generate interrupts when they cross a predefined threshold value. The Pentium Pro processor module itself provides a number of performance counters to count first- and second-level cache misses, as well as the frequencies of request types and the occupancies of internal resources. Together with the programmable handlers, these counters can provide a wealth of information about the behavior of the machine when running workloads.

8.7.7 Performance Characteristics

The memory bus within a quad has a peak bandwidth of 532 MB/s, and the SCI ring interconnect can transfer 500MB/s in each direction. The IQ-Link board can transfer data between these two at about 30MB/s in each direction (note that only a small fraction of the transactions appearing on the quad bus or on the SCI ring are expected to be relevant to the other). The latency for a local read miss satisfied in main memory (or the remote cache) is expected to be about 250ns under ideal conditions. The latency for a read satisfied in remote memory in a two-quad system is expected to be about 3.1 μs, a ratio of about 12 to 1. The latency through the data pump network interface for the first 18 bits of a transaction is 16ns, and then 2ns for every 18 bits thereafter. In the network itself, it takes about 26ns for the first bit to get from a quad into the Lash box that implements the ring and out to the data pump of the next quad along the ring.

The designers of the NUMA-Q have performed several experiments with microbenchmarks and with synthetic workloads that resemble database and transaction processing workloads. To obtain a flavor for the microbenchmark performance capabilities of the machine, how latencies vary under load, and the characteristics of these workloads, let us take a brief look at the results. Back-to-back read misses are found to take 600ns each and obtain a transfer bandwidth to the processor of 290 MB/s; back to back write misses take 525ns and sustain 185MB/s; and inbound writes from I/O devices to the local memory take 360ns at 88MB/s bandwidth, including bus arbitration.

Table 8-3 shows the latencies and characteristics "under load" as seen in actual workloads running on eight quads. The first two rows are for microbenchmarks designed to have all quads issuing read misses that are satisfied in remote memory. The third row is for a synthetic, on-line transaction processing benchmark, the fourth for a workload that looks somewhat like the Transaction Processing Council's TPC-C benchmark, and the last is for a decision support application

that issues very few remote read/write references since it is written using explicit message passing.

Table 8-3 Characteristics of microbenchmarks and workloads running on an 8-quad NUMA-Q.

Workload	Latency of L2 misses		SCLIC Utilization	Percentage of L2 misses satisfied in			
	All	Remotely satisfied		Local memory	Other Local cache	Local "remote cache"	Remote node
Remote Read Misses	7580ns	8170ns	93%	1%	0%	2%	97%
Remote Write Misses	8750ns	9460ns	94%	1%	0%	2%	97
OLTP-like	925ns	4700ns	62%	66%	20%	5%	9%
TPC-C like	875ns	6170ns	68%	73%	4%	16%	7%
Decision-support like	445ns	3120ns	9%	54.5%	40%	4%	1.5%

Remote latencies are significantly higher than the ideal for all but the message-passing decision support workload. In general, the SCI ring and protocol have higher latencies than those of more distributed networks and memory-based protocols (for example, requests that arrive late in a pending list have to wait a while to be satisfied). However, it turns out at least in these transaction-processing like workloads, much of the time in a remote access is spent passing through the IQ-Link board itself, and not in the bus or in the SCI protocol. Figure 8-32 shows the breakdowns of average remote latency into three components for two workloads on an 8-quad system. Clearly, the path to improved remote access performance, both under load and not under load, is to make the IQ-Link board more efficient. Opportunities being considered by the designers

[Note to artist: the 10 and 18 percent parts in the two pie-charts should be shaded similarly and captioned "SCI time", the 25 and 21 percent should be captioned "Bus time" and the 65 and 61 percent should be captioned "IQ-Link board time".]
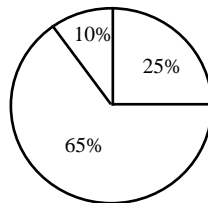
(a) TPC-C
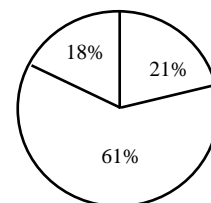
(a) TPC-D (decision support)

Figure 8-32 Components of average remote miss latency in two workloads on an 8-quad NUMA-Q.

include redesigning the SCLIC, perhaps using two instruction sequencers instead of one, and optimizing the OBIC, with the hope of reducing the remote access latency to about 2.5 μs under heavy load in the next generation.

8.7.8 Comparison Case Study: The HAL S1 Multiprocessor

The S1 multiprocessor from HAL Computer Systems is an interesting combination of some features of the NUMA-Q and the Origin2000. Like NUMA-Q it also uses Pentium Pro Quads as the processing nodes; however, it uses a memory-based directory protocol like that of the Origin2000 across Quads rather than the cache-based SCI protocol. Also, to reduce latency and assist occupancy, it integrates the coherence machinery more tightly with the node than the NUMA-Q does.

Instead of using separate chips for the directory protocol controller (SCLIC), bus interface controller (OBIC) and network interface (Data Pump), the S1 integrates the entire communication assist and the network interface into a single chip called the Mesh Coherence Unit (MCU).

Since the memory-based protocol does not require the use of forward and backward pointers with each cache entry, there is no need for a Quad-level remote data cache except to reduce capacity misses, and the S1 does not use a remote data cache. The directory information is maintained in separate SRAM chips, but the directory storage needed is greatly reduced by maintained information only for blocks that are in fact cached remotely, organizing the directory itself as a cache as will be discussed in Section 8.11.1. The MCU also contains a DMA engine to support explicit message passing and to allow transfers of large chunks of data through block data transfer even in the cache-coherent case. Message passing can be implemented either through the DMA engine (preferred for large messages) or through the transfer mechanism used for cache blocks (preferred for small messages). The MCU is hard-wired instead of programmable, which reduces its occupancy for protocol processing and hence improves its performance under contention. The MCU also has substantial hardware support for performance monitoring. The only other custom chip used is the network router, which is a six-ported crossbar with 1.9 million transistors, optimized for speed. The network is clocked at 200MHz. The latency through a single router is 42 ns, and the usable per-link bandwidth is 1.6GB/sec in each direction, similar to that of the Origin2000 network. The S1 interconnect implementation scales to 32 nodes (128 processors).

A major goal of integrating all the assist functionality into a single chip in S1 was to reduce remote access latency and increase remote bandwidth. From the designers' simulated measurements, the best-case unloaded latency for a read miss that is satisfied in local memory is 240ns, for a read miss to a block that is clean at a remote home is 1065 ns, and for a read miss to a block that is dirty in a third node is 1365 ns. The remote to local latency ratio is about 4-5, which is a little worse than on the SGI Origin2000 but better than on the NUMA-Q. The bandwidths achieved in copying a single 4KB page are 105 MB/s from local memory to local memory through processor reads and writes (limited primarily by the Quad memory controller which has to handle both the reads and writes of memory), about 70 MB/s between local memory and a remote memory (in either direction) when done through processor reads and writes, and about 270 MB/s in either direction between local and remote memory when done through the DMA engines in the MCUs. The case of remote transfers through processor reads and writes is limited primarily by the limit on the number of outstanding memory operations from a processor. The DMA case is also better because it requires only one bus transaction at the initiating end for each memory block, rather than two split transaction pairs in the case of processor reads and writes (once for the read and once for the write). At least in the absence of contention, the local Quad bus becomes a bandwidth bottleneck much before the interconnection network.

Having understood memory-based and cache-based protocols in some depth, let us now briefly examine some key interactions with the cost parameters of the communication architecture in determining performance.

8.8 Performance Parameters and Protocol Performance

Of the four major performance parameters in a communication architecture—overhead on the main processor, occupancy of the communication assist, network transit latency, and network bandwidth—the first is quite small on cache-coherent machines (unlike on message-passing sys-

tems, where it often dominates). In the best case, the portion that we can call processor overhead, and which cannot be hidden from the processor through overlap, it is the cost of issuing the memory operation. In the worst case, it is the cost of traversing the processor's cache hierarchy and reaching the assist. All other protocol processing actions are off-loaded to the communication assist (e.g. the Hub or the IQ-Link).

As we have seen, the communication assist has many roles in protocol processing, including generating a request, looking up the directory state, and sending out and receiving invalidations and acknowledgments. The occupancy of the assist for processing a transaction not only contributes to the uncontended latency of that transaction, but can also cause contention at the assist and hence increase the cost of other transactions. This is especially true in cache-coherent machines because there is a large number of small transactions—both data-carrying transactions and others like requests, invalidations and acknowledgments—so the occupancy is incurred very frequently and not amortized very well. In fact, because messages are small, assist occupancy very often dominates the node-to-network data transfer bandwidth as the key bottleneck to throughput at the end-points [HHS+95]. It is therefore very important to keep assist occupancy small. At the protocol level, it is important to not keep the assist tied up by an outstanding transaction while other transactions are available for it to process, and to reduce the amount of processing needed from the assist per transaction. For example, if the home forwards a request to a dirty node, the home assist should not be held up until the dirty node replies—dramatically increasing its effective occupancy—but should go on to service the next transaction and deal with the reply when it comes. At the design level, it is important to specialize the assist enough that its occupancy per transaction is low.

Impact of Network Latency and Assist Occupancy

Figure 8-33 shows the impact of assist occupancy and network latency on performance, assuming an efficient memory-based directory protocol similar to the one we shall discuss for the SGI Origin2000. In the absence of contention, assist occupancy behaves just like network transit latency or any other component of cost in a transaction's path. Increasing occupancy by x cycles would have the same impact as keeping occupancy constant but increasing transit latency by x cycles. Since the X-axis is total uncontended round-trip latency for a remote read miss—including the cost of transit latency and assist occupancies incurred along the way—if there is no contention induced by increasing occupancy then all the curves for different values of occupancy will be identical. In fact, they are not, and the separation of the curves indicates the impact of the contention induced by increasing assist occupancy.

The smallest value of occupancy (o) in the graphs is intended to represent that of an aggressive hard-wired controller such as the one used in the Origin2000. The least aggressive one represents placing a slow general-purpose processor on the memory bus to play the role of communication assist. The most aggressive transit latencies used represent modern high-end multiprocessor interconnects, while the least aggressive ones are closer to using system area networks like asynchronous transfer mode (ATM). We can see that for an aggressive occupancy, the latency curves take the expected $1/l$ shape. The contention induced by assist occupancy has a major impact on performance for applications that stress communication throughput (especially those in which communication is bursty), especially for the low-latency networks used in multiprocessors. For higher occupancies, the curve almost flattens with increasing latency. The problem is especially severe for applications with bursty communication, such as sorting and FFTs, since there the rate of communication relative to computation during the phase that performs communication does

not change much with problem size, so larger problem sizes do not help alleviate the contention during that phase. Assist occupancy is a less severe a problem for applications in which communication events are separated by computation and whose communication bandwidth demands are small (e.g. Barnes-Hut in the figure). When latency tolerance techniques are used (see Chapter 11), bandwidth is stressed even further so the impact of assist occupancy is much greater even at higher transit latencies [HHS+95]. These data show that is very important to keep assist occupancy low in machines that communicate and maintain coherence at a fine granularity such as that of cache blocks. The impact of contention due to assist occupancy tends to increase with

(a) FFT   (a) Ocean
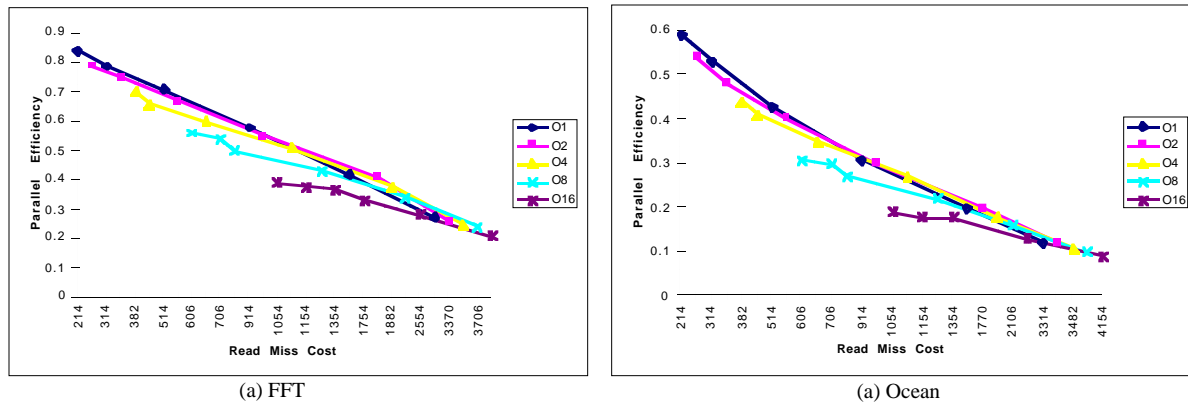
Figure 8-33 Impact of assist occupancy and network latency on the performance of memory-based cache coherence protocols.

The Y-axis is the parallel efficiency, which is the speedup over a sequential execution divided by the number of processors used (1 is ideal speedup). The X-axis is the uncontended round-trip latency of a read miss that is satisfied in main memory at the home, including all components of cost (occupancy, transit latency, time in buffers, and network bandwidth). Each curve is for a different value of assist occupancy (o), while along a curve the only parameter that is varies is the network transit latency (l). The lowest occupancy assumed is 7 processor cycles, which is labelled O1. O2 corresponds to twice that occupancy (14 processor cycles), and so on. All other costs, such as the time to propagate through the cache hierarchy and through buffers, and the node-to-network bandwidth are held constant. The graphs are for simulated 64-processor executions. The main conclusion is that the contention induced by assist occupancy is very important to performance, especially in low-latency networks.

the number of processors used to solve a given problem, since communication to computation ratio tends to increase.

Effects of Assist Occupancy on Protocol Tradeoffs

The occupancy of the assist has an impact not only on the performance of a given protocol but also on the tradeoffs among protocols. We have seen that cache-based protocols can have higher latency on write operations than memory-based protocols, since the transactions needed to invalidate sharers are serialized. The SCI cache-based protocol tends to have more protocol processing to do on a given memory operation than a memory-based protocol, so an assist for this protocol tends to have higher occupancy, so the effective occupancy of an assist tends to be significantly higher. Combined with the higher latency on writes, this would tend to cause memory-based protocols to perform better. This difference between the performance of the protocols will become greater as assist occupancy increases. On the other hand, the protocol processing occupancy for a given memory operation in SCI is distributed over more nodes and assists, so depending on the communication patterns of the application it may experience less contention at a given assist. For example, when hot-spotting becomes a problem due to bursty irregular communication in memory-based protocols (as in Radix sorting), it may be somewhat alleviated in SCI. SCI may also be more resilient to contention caused by data distribution problems or by less opti-

mized programs. How these tradeoffs actually play out in practice will depend on the characteristics of real programs and machines, although overall we might expect memory-based protocols to perform better in the most optimized cases.

Improving Performance Parameters in Hardware

There are many ways to use more aggressive, specialized hardware to improve performance characteristics such as latency, occupancy and bandwidth. Some notable techniques include the following. First, an SRAM directory cache may be placed close to the assist to reduce directory lookup cost, as is done in the Stanford FLASH multiprocessor [KOH+94]. Second, a single bit of SRAM can be maintained per memory block at the home, to keep track of whether or not the block is in clean state in the local memory. If it is, then on a read miss to a locally allocated block there is no need to invoke the communication assist at all. Third, if the assist occupancy is high, it can be pipelined into stages of protocol processing (e.g. decoding a request, looking up the directory, generating a response) or its occupancy can be overlapped with other actions. The former technique reduces contention but not the uncontended latency of memory operations, while the latter does the reverse. Critical path latency for an operation can also be reduced by having the assist generate and send out a response or a forwarded request even before all the cleanup it needs to do is done.

This concludes our discussion of directory-based cache coherence protocols. Let us now look at synchronization on scalable cache-coherent machines.

8.9 Synchronization

Software algorithms for synchronization on scalable shared address space systems using atomic exchange instructions or LL-SC have already been discussed in Chapter 7 (Section 7.10). Recall that the major focus of these algorithms compared to those for bus-based machines was to exploit the parallelism of paths in the interconnect and to ensure that processors would spin on local rather than nonlocal variables. The same algorithms are applicable to scalable cache-coherent machines. However, there are two differences. On one hand, the performance implications of spinning on remotely allocated variables are likely to be much less significant, since a processor caches the variable and then spins on it locally until it is invalidated. Having processors spin on different variables rather than the same one, as achieved in the tree barrier, is of course useful so that not all processors rush out to the same home memory when the variable is written and invalidated, reducing contention. However, there is only one (very unlikely) reason that it may actually be very important to performance that the variable a processor spins on is allocated locally: if the cache is unified and direct-mapped and the instructions for the spin loop conflict with the variable itself, in which case conflict misses will be satisfied locally. A more minor benefit of good placement is converting the misses that occur after invalidation into 2-hop misses from 3-hop misses. On the other hand, implementing atomic primitives and LL-SC is more interesting when it interacts with a coherence protocol. This section examines these two aspects, first comparing the performance of the different synchronization algorithms for locks and barriers that were described in Chapters 5 and 7 on the SGI Origin2000, and then discussing some new implementation issues for atomic primitives beyond the issues already encountered on bus-based machines.

8.9.1 Performance of Synchronization Algorithms

The experiments we use to illustrate synchronization performance are the same as those we used on the bus-based SGI Challenge in Section 5.6, again using LL-SC as the primitive used to construct atomic operations. The delays used are the same in processor cycles, and therefore different in actual microseconds. The results for the lock algorithms described in Chapters 5 and 7 are shown in Figure 8-34 for 16-processor executions. Here again, we use three different sets of values for the delays within and after the critical section for which processors repeatedly contend.

(a) null (c=0, d=0)　　　　(b) critical-section (c=3.64 μs, d=0)　　　　(c) delay (c=3.64 μs, d=1.29 μs)
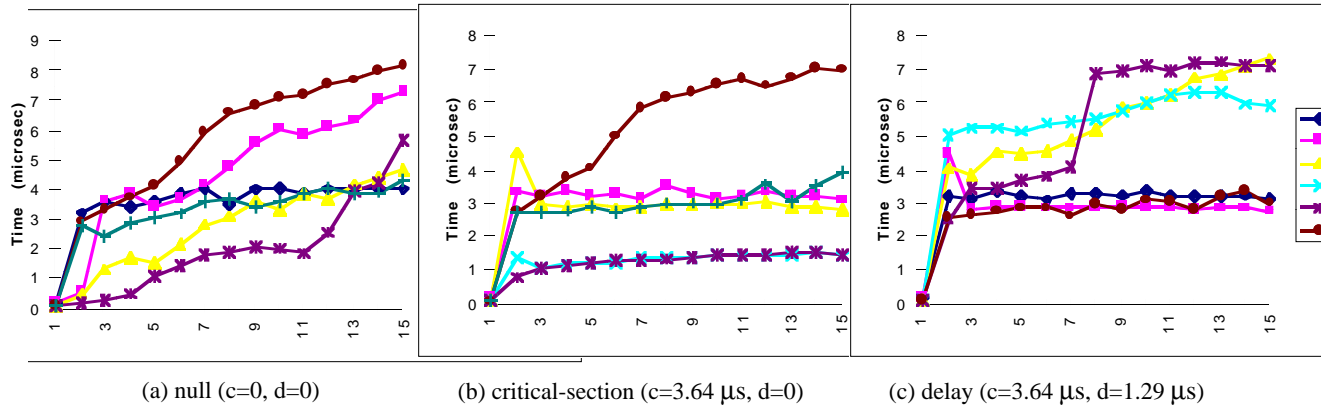
Figure 8-34 Performance of locks on the SGI Origin2000, for three different scenarios.

Here too, until we use delays between critical sections the simple locks behavior unfairly and yield higher throughput. Exponential backoff helps the LL-SC when there is a null critical section, since this is the case where there is significant contention to be alleviated. Otherwise, with null critical sections the ticket lock itself scales quite poorly as before, but scales very well when proportional backoff is used. The array-based lock also scales very well. With coherent caches, the better placement of lock variables in main memory afforded by the software queuing lock is not particularly useful, and in fact the queueing lock incurs contention on its compare and swap operations (implemented with LL-SC) and scales worse than the array lock. If we force the simple locks to behave fairly, they behave much like the ticket lock without proportional backoff.

If we use a non-null critical section and a delay between lock accesses (Figure 8-34(c)), all locks behave fairly. Now, the simple LL-SC locks don't have their advantage, and their scaling disadvantage shows through. The array-based lock, the queueing lock, and the ticket lock with proportional backoff all scale very well. The better data placement of the queuing lock does not matter, but neither is the contention any worse for it now. Overall, we can conclude that the array-based lock and the ticket lock perform well and robustly for scalable cache-coherent machines, at least when implemented with LL-SC, and the simple LL-SC lock with exponential backoff performs best when there is no delay between an unlock and the next lock, due to repeated unfair successful access by a processor in its own cache. The most sophisticated, queueing lock is unnecessary, but also performs well when there are delays between unlock and lock.

8.9.2 Supporting Atomic Primitives

Consider implementing atomic exchange (read-modify-write) primitives like test&set performed on a memory location. What matters here is that a conflicting write *to that location* by another processor occur either before the read component of the read-modify-write operation or after its write component. As we discussed for bus-based machines in Chapter 5 (Section 5.6.3), the read component may be allowed to complete as soon as the write component is serialized with respect to other writes. and we can ensure that no incoming invalidations are applied to the block until the read has completed. If the read-modify-write is implemented at the processor (cacheable primitives), this means that the read can complete once the write has obtained ownership, and even before invalidation acknowledgments have returned. Atomic operations can also be implemented at the memory, but it is easier to do this if we disallow the block from being cached in dirty state by any processor. Then, all writes go to memory, and the read-modify-write can be serialized with respect to other writes as soon as it gets to memory. Memory can send a response to the read component in parallel with sending out invalidations corresponding to the write component.

Implementing LL-SC requires all the same consideration to avoid livelock as it did for bus-based machines, plus one further complication. Recall that an SC should not send out invalidations/ updates if it fails, since otherwise two processors may keep invalidating/updating each other and failing, causing livelock. To detect failure, the requesting processor needs to determine if some other processor's write to the block has happened before the SC. In a bus-based system, the cache controller can do this by checking upon an SC whether the cache no longer has a valid copy of the block, or whether there are incoming invalidations/updates for the block that have already appeared on the bus. The latter detection, which basically checks the serialization order, cannot be done locally by the cache controller with a distributed interconnect. In an invalidation-based protocol, if the block is still in valid state in the cache then the read exclusive request corresponding to the SC goes to the directory at the home. There, it checks to see if the requestor is still on the sharing list. If it isn't, then the directory knows that another conflicting write has been serialized before the SC, so it does not send out invalidations corresponding to the SC and the SC fails. In an update protocol, this is more difficult since even if another write has been serialized before the SC the SC requestor will still be on the sharing list. One solution [Gha95] is to again use a two-phase protocol. When the SC reaches the directory, it locks down the entry for that block so no other requests can access it. Then, the directory sends a message back to the SC requestor, which upon receipt checks to see if the lock flag for the LL-SC has been cleared (by an update that arrived between now and the time the SC request was sent out). If so, the SC has failed and a message is sent back to the directory to this effect (and to unlock the directory entry). If not, then as long as there is point to point order in the network we can conclude that no write beat the SC to the directory so the SC should succeed. The requestor sends this acknowledgment back to the directory, which unlocks the directory entry and sends out the updates corresponding to the SC, and the SC succeeds.

8.10 Implications for Parallel Software

What distinguishes the coherent shared address space systems described in this chapter from those described in Chapters 5 and 6 is that they all have physically distributed rather than centralized main memory. Distributed memory is at once an opportunity to improve performance through data locality and a burden on software to exploit this locality. As we saw in Chapter 3, in

the cache-coherent machine with physically distributed memory (or CC-NUMA machines) discussed in this chapter, parallel programs may need to be aware of physically distributed memory, particularly when their important working sets don't fit in the cache. Artifactual communication occurs when data are not allocated in the memory of a node that incurs capacity, conflict or cold misses on them (it can also occur due to false sharing or fragmentation of communication, which occur at cache block granularity). Consider also a multiprogrammed workload, consisting of even sequential programs, in which application processes are migrated among processing nodes for load balancing. Migrating a process will turn what should be local misses into remote misses, unless the system moves all the migrated process's data to the new node's main memory as well.

In the CC-NUMA machines discussed in this chapter, main memory management is typically done at the fairly large granularity of pages. The large granularity can make it difficult to distribute shared data structures appropriately, since data that should be allocated on two different nodes may fall on the same unit of allocation. The operating system may migrate pages to the nodes that incur cache misses on them most often, using information obtained from hardware counters, or the runtime system of a programming language may do this based on user-supplied hints or compiler analysis. More commonly today, the programmer may direct the operating system to place pages in the memories closest to particular processes. This may be as simple as providing these directives to the system—such as "place the pages in this range of virtual addresses in this process X's local memory"—or it may involve padding and aligning data structures to page boundaries so they can be placed properly, or it may even require that data structures be organized differently to allow such placement at page granularity. We saw examples in using four-dimensional instead of two-dimensional arrays in the equation solver kernel, in Ocean and in LU. Simple, regular cases like these may also be handled by sophisticated compilers. In Barnes-Hut, on the other hand, proper placement would require a significant reorganization of data structures as well as code. Instead of having a single linear array of particles (or cells), each process would have an array or list of its own assigned particles that it could allocate in its local memory, and between time-steps particles that were reassigned would be moved from one array/ list to another. However, as we have seen data placement is not very useful for this application, and may even hurt performance due to its high costs. It is important that we understand the costs and potential benefits of data migration. Similar issues hold for software-controlled replication, and the next chapter will discuss alternative approaches to coherent replication and migration in main memory.

One of the most difficult and insidious problems for a programmer to deal with in a coherent shared address space is contention. Not only is data traffic implicit and often unpredictable, but contention can also be caused by "invisible" protocol transactions such as the ownership requests above, invalidations and acknowledgments, which a programmer is not inclined to think about at all. All of these types of transactions occupy the protocol processing portion of the communication assist, so it is very important to keep the occupancy of the assist per transaction very low to contain end-point contention. Invisible protocol messages and contention make performance problems like false sharing all the more important for a programmer to avoid, particularly when they cause a lot of protocol transactions to be directed toward the same node. For example, we are often tempted to structure some kinds of data as an array with one entry per process. If the entries are smaller than a page, several of them will fall on the same page. If these array entries are not padded to avoid false sharing, or if they incur conflict misses in the cache, all the misses and traffic will be directed at the home of that page, causing a lot of contention. If we do structure data this way, then in a distributed memory machine it is advantageous not only to structure multiple such arrays as an array of records, as we did to avoid false sharing in Chapter 5, but also to pad and align the records to a page and place the pages in the appropriate local memories.

An interesting example of how contention can cause different orchestration strategies to be used in message passing and shared address space systems is illustrated by a high-performance parallel Fast Fourier Transform. Conceptually, the computation is structured in phases. Phases of local computation are separated by phases of communication, which involve the transposition of a matrix. A process reads columns from a source matrix and writes them into its assigned rows of a destination matrix, and then performs local computation on its assigned rows of the destination matrix. In a message passing system, it is important to coalesce data into large messages, so it is necessary for performance to structure the communication this way (as a separate phase apart from computation). However, in a cache-coherent shared address space there are two differences. First, transfers are always done at cache block granularity. Second, each fine-grained transfer involves invalidations and acknowledgments (each local block that a process writes is likely to be in shared state in the cache of another processor from a previous phase, so must be invalidated), which cause contention at the coherence controller. It may therefore be preferable to perform the communication on-demand at fine grain while the computation is in progress, thus staggering out the communication and easing the contention on the controller. A process that computes using a row of the destination matrix can read the words of that source matrix column from a remote node on-demand while it is computing, without having to do a separate transpose phase.

Finally, synchronization can be expensive in scalable systems, so programs should make a special effort to reduce the frequency of high-contention locks or global barrier synchronization.

8.11 Advanced Topics

Before concluding the chapter, this section covers two additional topics. The first is the actual techniques used to reduce directory storage overhead in flat, memory-based schemes, as promised in Section 8.3.3. The second is techniques for hierarchical coherence, both snooping- and directory-based.

8.11.1 Reducing Directory Storage Overhead

In the discussion of flat, memory based directories in Section 8.3.3, it was stated that the size or width of a directory entry can be reduced by using a limited number of pointers rather than a full bit-vector, and that doing this requires some overflow mechanism when the number of copies of the block exceeds the number of available pointers. Based on the empirical data about sharing patterns, the number of pointers likely to be provided in limited pointer directories is very small, so it is important that the overflow mechanism be efficient. This section first discusses some possible overflow methods for limited pointer directories. It then examines techniques to reduce directory "height", by not having a directory entry for every memory block in the system but rather organizing the directory as a cache. The overflow methods include schemes called broadcast, no broadcast, coarse vector, software overflow, and dynamic pointers.

Broadcast (Dir$_i$B) The overflow strategy in the Dir$_i$B scheme [ASH+88] is to set a *broadcast bit* in the directory entry when the number of available pointers (indicated by subscript i in Dir$_i$B) is exceeded. When that block is written again, invalidation messages are sent to all nodes, regardless of whether or not they were caching the block. It is not semantically incorrect, but simply wasteful, to send an invalidation message to a processor not caching the block. Network band-

width may be wasted, and latency stalls may be increased if the processor performing the write must wait for acknowledgments before proceeding. However, the method is very simple.

No broadcast (Dir$_i$NB) The Dir$_i$NB scheme [ASH+88] avoids broadcast by never allowing the number of valid copies of a block to exceed *i*. Whenever the number of sharers is *i* and another node requests a read-shared copy of the block, the protocol invalidates the copy in one of the existing sharers and frees up that pointer in the directory entry for the new requestor. A major drawback of this scheme is that it does not deal well with data that are read by many processors during a period, even though the data do not have to be invalidated. For example, a piece of code that is running on all the processors, or a small table of pre-computed values shared by all processors will not be able to reside in all the processors' caches at the same time, and a continual stream of misses will be generated. While special provisions can be made for blocks containing code, for example, their consistency may be managed by software instead of hardware, it is not clear how to handle widely shared read-mostly data well in this scheme.

Coarse vector (Dir$_i$CV$_r$) The Dir$_i$CV$_r$ scheme [GWM90] also uses *i* pointers in its initial representation, but overflows to a coarse bit vector scheme like the one used by the Origin2000. In this representation, each bit of the directory entry is used to point to a unique group of the nodes in the machine (the subscript *r* in Dir$_i$CV$_r$ indicates the size of the group), and that bit is turned ON whenever *any* node in that partition is caching that block. When a processor writes that block, all

(a) No overflow   (a) Overflow
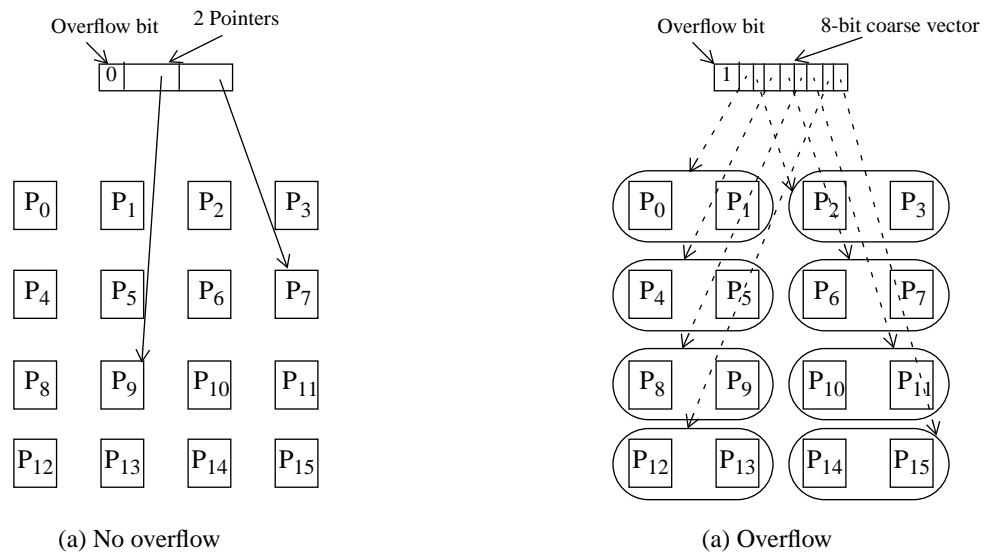
Figure 8-35 The change in representation in going from limited pointer representation to coarse vector representation on overflow.

Upon overflow, the two 4-bit pointers (for a 16-node system) are viewed as an eight-bit coarse vector, each bit corresponding to a group of two nodes. The overflow bit is also set, so the nature of the representation can be easily determined. The dotted lines in (b) indicate the correspondence between bits and node groups.

nodes in the groups whose bits are turned ON are sent an invalidation message, regardless of whether they have actually accessed or are caching the block. As an example, consider a 256-node machine for which we store eight pointers in the directory entry. Since each pointer needs to be 8-bits wide, there are 64-bits available for the coarse-vector on overflow. Thus we can implement a Dir$_8$CV$_4$ scheme, with each coarse-vector bit pointing to a group of four nodes. A single bit per entry keeps track of whether the current representation is the normal limited pointers or a

coarse vector. As shown in Figure 8-36, an advantage of a scheme like Dir_iCV_r (and several of the following) over Dir_iB and Dir_iNB is that its behavior is more robust to different sharing patterns.
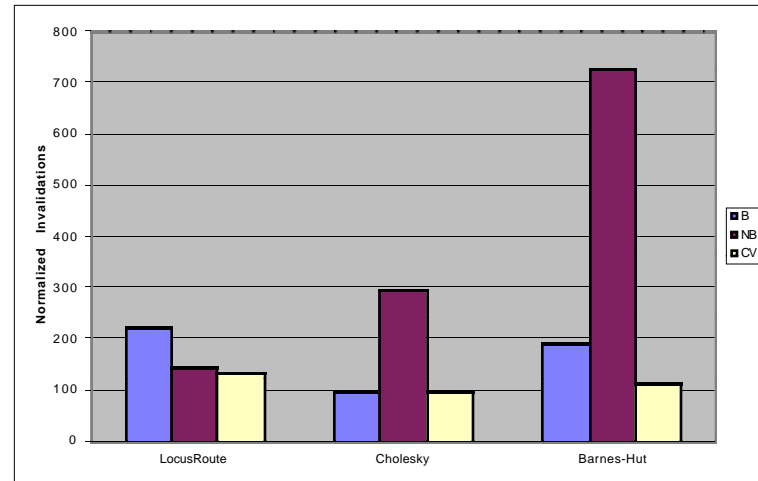
Figure 8-36 Comparison of invalidation traffic generated by Dir_4B, Dir_4NB, and Dir_4CV_4 schemes as compared to the full-bit-vector scheme.

The results are taken from [Web93], so the simulation parameters are different from those used in this book. The number of processors (one per node) is 64. The data for LocusRoute application, which has data that are written with some frequency and read by many nodes, shows the potential pitfalls of the Dir_iB scheme. Cholesky and Barnes-Hut, which have data that are shared by large numbers of processors (e.g., nodes close to root of tree in Barnes-Hut) show the potential pitfalls of Dir_iNB scheme. The Dir_iCV_r scheme is found to be reasonably robust.

Software Overflow (Dir_iSW) This scheme is different from the previous ones in that it does not throw away the precise caching status of a block when overflow occurs. Rather, the current i pointers and a pointer to the new sharer are saved into a special portion of the node's local main memory by software. This frees up space for new pointers, so i new sharers can be handled by hardware before software must be invoked to store pointers away into memory again. The overflow also causes an overflow bit to be set in hardware. This bit ensures that when a subsequent write is encountered the pointers that were stored away in memory will be read out and invalidation messages sent to those nodes as well. In the absence of a very sophisticated (programmable) communication assist, the overflow situations (both when pointers must be stored into memory and when they must be read out and invalidations sent) are handled by software running on the main processor, so the processor must be interrupted upon these events. The advantages of this scheme are that precise information is kept about sharers even upon overflow, so there is no extra traffic generated, and the overflow handling is managed by software. The major overhead is the cost of the interrupts and software processing. This disadvantage takes three forms: (i) the processor at the home of the block spends time handling the interrupt instead of performing user's computation, (ii) the overhead for performing these requests is large, thus potentially becoming a bottleneck for contention if many requests come to it at the same time, and (iii) the requesting processor may stall longer due to the higher latency of the requests that cause an interrupt as well as increased contention.[1]

Example 8-3   Software overflow for limited pointer directories was used in the MIT Alewife research prototype [ABC+95] and called the "LimitLESS" scheme [ALK+91]. The

Alewife machine is designed to scale to 512 processors. Each directory entry is 64-bits wide. It contains five 9-bit pointers to record remote processors caching the block, and one dedicated bit to indicate whether the local processor is also caching the block (thus saving 8 bits when this is true). Overflow pointers are stored in a hash-table in the main memory. The main processor in Alewife has hardware support for multithreading (see Chapter 11), with support for fast handing of traps upon overflow. Nonetheless, while the latency of a request that causes 5 invalidations and can be handled in hardware is only 84 cycles, a request requiring 6 invalidations and hence software intervention takes 707 cycles.

Dynamic pointers (Dir$_i$DP) The Dir$_i$DP scheme [SiH91] is a variation of the Dir$_i$SW scheme. In this scheme, in addition to the *i* hardware pointers each directory entry contains a pointer into a special portion of the local node's main memory. This special memory has a free list associated with it, from which pointer-structures can be dynamically allocated to processors as needed. The key difference from Dir$_i$SW is that all linked-list manipulation is done by a special-purpose protocol hardware/processor rather than by the general-purpose processor of the local node. As a result, the overhead of manipulating the linked-lists is small. Because it also contains a pointer to memory, the number of hardware pointers *i* used in this scheme is typically very small (1 or 2). The Dir$_i$DP scheme is being used as the default directory organization for the Stanford FLASH multiprocessor [KOH+94].

Among these many alternative schemes for maintaining directory information in a memory-based protocol, it is quite clear that the Dir$_i$B and Dir$_i$NB schemes are not very robust to different sharing patterns. However, the actual performance (and cost/performance) tradeoffs among the others are not very well understood for real applications on large-scale machines. The general consensus seems to be that full bit vectors are appropriate for machines with a moderate number of processing nodes, while several among the last few schemes above ought to suffice for larger systems.

In addition to reducing directory entry width, an orthogonal way to reduce directory memory overhead is to reduce the total number of directory entries used by not using one per memory block [GWM90, OKN90]; that is, to go after the *M* term in the *P*M* expression for directory memory overhead. Since the two methods of reducing overhead are orthogonal, they can be traded off against each other: Reducing the number of entries allows us to make entries wider (use more hardware pointers) without increasing cost, and vice versa.

The observation that motivates the use of fewer directory entries is that the total amount of cache memory is much less than the total main memory in the machine. This means that only a very small fraction of the memory blocks will be cached at a given time. For example, each processing node may have a 1 Mbyte cache and 64 Mbytes of main memory associated with it. If there were one directory entry per memory block, then across the whole machine 63/64 or 98.5% of the directory entries will correspond to memory blocks that are not cached anywhere in the machine. That is an awful lot of directory entries lying idle with no bits turned ON, especially if replacement hints are used. This waste of memory can be avoided by organizing the directory as a cache,

---

1. It is actually possible to reply to the requestor before the trap is handled, and thus not affect the latency seen by it. However, that simply means that the next processor's request is delayed and that processor may experience a stall.

and allocating the entries in it to memory blocks dynamically just as cache blocks are allocated to memory blocks containing program data. In fact, if the number of entries in this directory cache is small enough, it may enable us to use fast SRAMs instead of slower DRAMs for directories, thus reducing the access time to directory information. This access time is in the critical path that determines the latency seen by the processor for many types of memory references. Such a directory organization is called a *sparse directory*, for obvious reasons.

While a sparse directory operates quite like a regular processor cache, there are some significant differences. First, there is no need for a backing store for this cache: When an entry is replaced, if any node's bits in it are turned on then we can simply send invalidations or flush messages to those nodes. Second, there is only one directory entry per cache entry, so spatial locality is not an issue. Third, a sparse directory handles references from potentially all processors, whereas a processor cache is only accessed by the processor(s) attached to it. And finally, the references stream the sparse directory sees is heavily filtered, consisting of only those references that were not satisfied in the processor caches. For a sparse directory to not become a bottleneck, it is essential that it be large enough and have enough associativity to not incur too many replacements of actively accessed blocks. Some experiments and analysis studying the sizing of the directory cache can be found in [Web93].

8.11.2 Hierarchical Coherence

In the introduction to this chapter, it was mentioned that one way to build scalable coherent machines is to hierarchically extend the snoopy coherence protocols based on buses and rings that we have discussed in Chapters 5 and 6 in this chapter, respectively. We have also been introduced to hierarchical directory schemes in this chapter. This section describes hierarchical approaches to coherence a little further. While hierarchical ring-based snooping has been used in commercial systems (e.g. in the Kendall Square Research KSR-1 [FBR93]) as well as research prototypes (e.g. the University of Toronto's Hector system [VSL+91, FVS92]), and hierarchical directories have been studied in academic research, these approaches have not gained much favor. Nonetheless, building large systems hierarchically out of smaller ones is an attractive abstraction, and it is useful to understand the basic techniques.

Hierarchical Snooping

The issues in hierarchical snooping are similar for buses and rings, so we study them mainly through the former. A bus hierarchy is a tree of buses. The leaves are bus-based multiprocessors that contain the processors. The busses that constitute the internal nodes don't contain processors, but are used for interconnection and coherence control: They allow transactions to be snooped and propagated up and down the hierarchy as necessary. Hierarchical machines can be built with main memory centralized at the root or distributed among the leaf multiprocessors (see Figure 8-37). While a centralized main memory may simplify programming, distributed memory has advantages both in bandwidth and potentially in performance (note however that if data are not distributed such most cache misses are satisfied locally, remote data may actually be further away than the root in the worst case, potentially leading to worse performance). Also, with distributed memory, a leaf in the hierarchy is a complete bus-based multiprocessor, which is already a commodity product with cost advantages. Let us focus on hierarchies with distributed memory, leaving centralized memory hierarchies to the exercises.

The processor caches within a leaf node (multiprocessor) are kept coherent by any of the snoopy

(a) Centralized Memory

(b) Distributed Memory
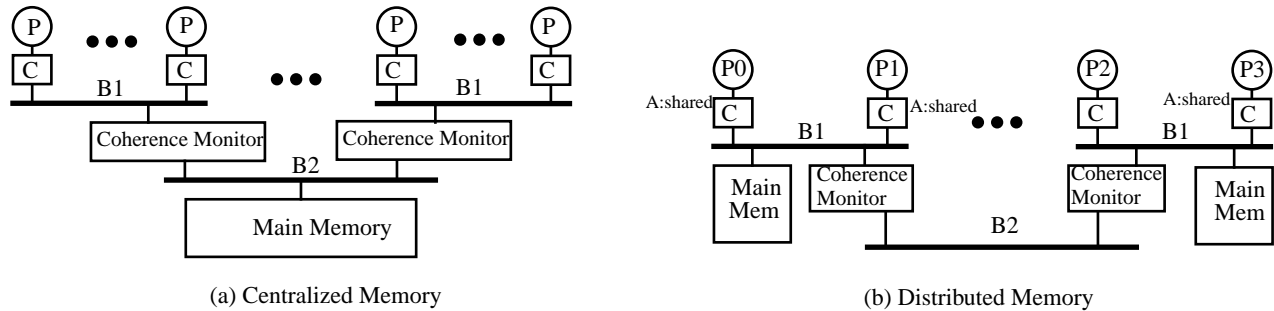
Figure 8-37 Hierarchical bus-based multiprocessors, shown with a two-level hierarchy.

protocols discussed in Chapter 5. In a simple, two-level hierarchy, we connect several of these bus-based systems together using another bus (B2) that has the main memory attached to it (the extension to multilevel hierarchies is straightforward). What we need is a coherency monitor associated with each B1 bus that monitors (snoops) the transactions on both buses, and decides which transactions on its B1 bus should be forwarded to the B2 bus and which ones that appear on the B2 bus should be forwarded to its B1 bus. This device acts as a filter, forwarding only the necessary transactions in both directions, and thus reduces the bandwidth demands on the buses.

In a system with distributed memory, the coherence monitor for a node has to worry about two types of data for which transactions may appear on either the B1 or B2 bus: data that are allocated nonlocally but cached by some processor in the local node, and data that are allocated locally but cached remotely. To watch for the former on the B2 bus, a remote access cache per node can be used as in the Sequent NUMA-Q. This cache maintains inclusion (see Section 6.4.1) with regard to remote data cached in any of the processor caches on that node, including a dirty-but-stale bit indicating when a processor cache in the node has the block dirty. Thus, it has enough information to determine which transactions are relevant in each direction and pass them along.

For locally allocated data, bus transactions can be handled entirely by the local memory or caches, except when the data are cached by processors in other (remote) nodes. For these data, there is no need to keep the data themselves in the coherence monitor, since either the valid data are already available locally or they are in modified state remotely; in fact, we would not want to since the amount of data may be as large as the local memory. However, the monitor keeps state information for locally allocated blocks that are cached remotely, and snoops the local B1 bus so relevant transactions for these data can be forwarded to the B2 bus if necessary. Let's call this part of the coherence monitor the local state monitor. Finally, the coherence monitor also watches the B2 bus for transactions to its local addresses, and passes them on to the local B1 bus unless the local state monitor says they are cached remotely in a modified state. Both the remote cache and the local state part of the monitor are looked up on B1 and B2 bus transactions.

Consider the coherence protocol issues (a) through (c) that were raised in Section 8.2. (a) Information about the state in other nodes (subtrees) is implicitly available to enough of an extent in the local coherence monitor (remote cache and local state monitor); (b) if this information indicates a need to find other copies beyond the local node, the request is broadcast on the next bus (and so on hierarchically in deeper hierarchies), and other relevant monitors will respond; and (c)

communication with the other copies is performed simultaneously as part of finding them through the hierarchical broadcasts on buses.

More specifically, let us consider the path of a read miss, assuming a shared physical address space. A BusRd request appears on the local B1 bus. If the remote access cache, the local memory or another local cache has a valid copy of the block, they will supply the data. Otherwise, either the remote cache or the local state monitor will know to pass the request on to the B2 bus. When the request appears on B2, the coherence monitors of other nodes will snoop it. If a node's local state monitor determines that a valid copy of the data exists in the local node, it will pass the request on to the B1 bus, wait for the reply and put it back on the B2 bus. If a node's remote cache contains the data, then if it has it in shared state it will simply reply to the request itself, if in dirty state it will reply and also broadcast a read on the local bus to have the dirty processor cache downgrade the block to shared, and if dirty-but-stale then it will simply broadcast the request on the local bus and reply with the result. In the last case, the local cache that has the data dirty will change its state from dirty to shared and put the data on the B1 bus. The remote cache will accept the data reply from the B1 bus, change its state from dirty-but-stale to shared, and pass the reply on to the B2 bus. When the data response appears on B2, the requestor's coherence monitor picks it up, installs it and changes state in the remote cache if appropriate, and places it on its local B1 bus. (If the block has to be installed in the remote cache, it may replace some other block, which will trigger a flush/invalidation request on that B1 bus to ensure the inclusion property.) Finally, the requesting cache picks up the response to its BusRd request from the B1 bus and stores it in shared state.

For writes, consider the specific situation shown in Figure 8-37(b), with P0 in the left hand node issuing a write to location A, which is allocated in the memory of a third node (not shown). Since P0's own cache has the data only in shared state, an ownership request (BusUpgr) is issued on the local B1 bus. As a result, the copy of B in P1's cache is invalidated. Since the block is not available in the remote cache in dirty-but-stale state (which would have been incorrect since P1 had it in shared state), the monitor passes the BusUpgr request to bus B2, to invalidate any other copies, and at the same time updates its own state for the block to dirty-but-stale. In another node, P2 and P3 have the block in their caches in shared state. Because of the inclusion property, their associated remote cache is also guaranteed to have the block in shared state. This remote cache passes the BusUpgr request from B2 onto its local B1 bus, and invalidates its own copy. When the request appears on the B1 bus, the copies of B in P2 and P3's caches are invalidated. If there is another node on the B2 bus whose processors are not caching block B, the upgrade request will not pass onto its B1 bus. Now suppose another processor P4 in the left hand node issues a store to location B. This request will be satisfied within the local node, with P0's cache supplying the data and the remote cache retaining the data in dirty-but-stale state, and no transaction will be passed onto the B2 bus.

The implementation requirements on the processor caches and cache controllers remain unchanged from those discussed in Chapter 6. However, there are some constraints on the remote access cache. It should be larger than the sum of the processor caches and quite associative, to maintain inclusion without excessive replacements. It should also be *lockup-free*, i.e. able to handle multiple requests from the local node at a time while some are still outstanding (more on this in Chapter 11). Finally, whenever a block is replaced from the remote cache, an invalidate or flush request must be issued on the B1 bus depending on the state of the replaced block (shared or dirty-but-stale, respectively). Minimizing the access time for the remote cache is less critical than increasing its hit-rate, since it is not in the critical path that determines the clock-rate of the processor: The remote caches are therefore more likely to be built out of DRAM rather than SRAM.

The remote cache controller must also deal with non-atomicity issues in requesting and acquiring buses that are similar to the ones we discussed in Chapter 6.

Finally, what about write serialization and determining store completion? From our earlier discussion of how these work on a single bus in Chapter 6, it should be clear that serialization among requests from different nodes will be determined by the order in which those requests appear on the closest bus to the root on which they both appear. For writes that are satisfied entirely within the same leaf node, the order in which they may be seen by other processors—within or without that leaf—is their serialization order provided by the local B1 bus. Similarly, for writes that are satisfied entirely within the same subtree, the order in which they are seen by other processors—within or without that subtree—is the serialization order determined by the root bus of that subtree. It is easy to see this if we view each bus hanging off a shared bus as a processor, and recursively use the same reasoning that we did with a single bus in Chapter 5. Similarly, for store completion, a processor cannot assume its store has committed until it appears on the closest bus to the root on which it will appear. An acknowledgment cannot be generated until that time, and even then the appropriate orders must be preserved between this acknowledgment and other transactions on the way back to the processor (see Exercise 8.11). The request reaching this bus is the point of commitment for the store: once the acknowledgment of this event is sent back, the invalidations themselves no longer need to be acknowledged as they make their way down toward the processor caches, as long as the appropriate orders are maintained along this path (just as with multi-level cache hierarchies in Chapter 6).

Example 8-4   **Encore GigaMax** One of the earliest machines that used the approach of hierarchical snoopy buses with distributed memory was the Gigamax [Wil87, WWS+89] from Encore Corporation. The system consisted of up to eight Encore Multimax machines (each a regular snoopy bus-based multiprocessor) connected together by fiber-optic links to a ninth global bus, forming a two-level hierarchy. Figure 8-38 shows a block diagram. Each node is augmented with a Uniform Interconnection Card (UIC) and a Uniform Cluster (Node) Cache (UCC) card. The UCC is the remote access cache, and the UIC is the local state monitor. The monitoring of the global bus is done differently in the Gigamax due to its particular organization. Nodes are connected to the global bus through a fiber-optic link, so while the local node cache (UCC) caches remote data it does not snoop the global bus directly. Rather, every node also has a corresponding UIC on the global bus, which monitors global bus transactions for remote memory blocks that are cached in this local node. It then passes on the relevant requests to the local bus. If the UCC indeed sat directly on the global bus as well, the UIC on the global bus would not be necessary. The reason the GigaMax use fiber-optic links and not a single UIC sitting on both buses is that high-speed buses are usually short: The Nanobus used in the Encore Multimax and GigaMax is 1 foot long (light travels 1 foot in a nanosecond, hence the name Nanobus). Since each node is at least 1 foot wide, and the global bus is also 1 foot wide, flexible cabling is needed to hook these together. With fiber, links can be made quite long without affecting their transmission capabilities.

The extension of snoopy cache coherence to hierarchies of rings is much like the extension of hierarchies of buses with distributed memory. Figure 8-39 shows a block diagram. The local rings and the associated processors constitute nodes, and these are connected by one or more glo-
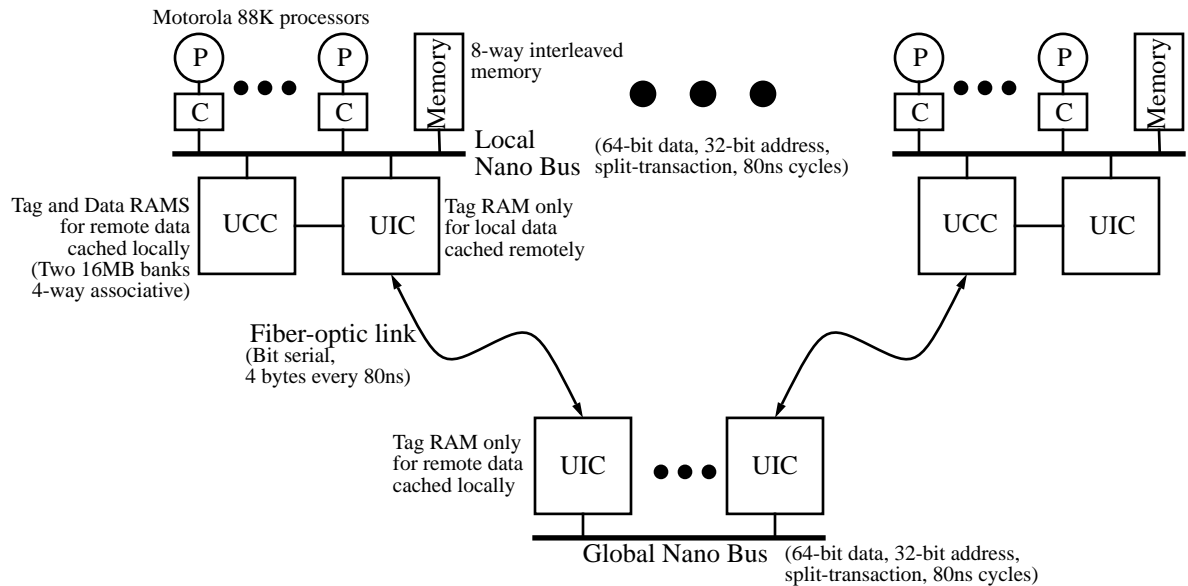
Figure 8-38 Block diagram for the Encore Gigamax multiprocessor.
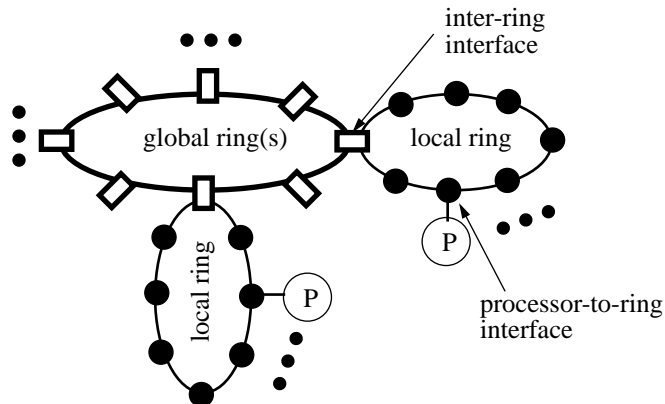
Figure 8-39 Block diagram for a hierarchical ring-based multiprocessor.

bal rings. The coherence monitor takes the form of an inter-ring interface, serving the same roles as the coherence monitor (remote cache and local state monitor) in a bus hierarchy.

Hierarchical Directory Schemes

Hierarchy can also be used to construct directory schemes. Here, unlike in flat directory schemes, the source of the directory information is not found by going to a fixed node. The locations of copies are found neither at a fixed home node nor by traversing a distributed list pointed to by that home. And invalidation messages are not sent directly to the nodes with copies. Rather, all these activities are performed by sending messages up and down a hierarchy (tree) built upon the nodes, with the only direct communication (network transactions) being between parents and children in the tree.

At first blush the organization of hierarchical directories is much like hierarchical snooping. Consider the example shown in Figure 8-40. The processing nodes are at the leaves of the tree and

processing nodes

(Tracks which of its children
processing nodes have a copy
of the memory block. Also tracks
which local memory blocks are
cached outside this subtree.
Inclusion is maintained between
processor caches and directory.)

level-1 directory

level-2 directory

(Tracks which of its children
level-1 directories have a copy
of the memory block. Also tracks
which local memory blocks are
cached outside this subtree.
Inclusion is maintained between
level-1 directories and level-2 directory.)

Figure 8-40 Organization of hierarchical directories.

main memory is distributed along with the processing nodes. Every block has a home memory (leaf) in which it is allocated, but this does not mean that the directory information is maintained or rooted there. The internal nodes of the tree are not processing nodes, but only hold directory information. Each such directory node keeps track of all memory blocks that are being cached/recorded by its immediate sub-trees. It uses a presence vector per block to tell which of its sub-trees have copies of the block, and a bit to tell whether one of them has it dirty. It also records information about local memory blocks (i.e., blocks allocated in the local memory of one of its children) that are being cached by processing nodes outside its subtree. As with hierarchical snooping, this information is used to decide when requests originating within the subtree should be propagated further up the hierarchy. Since the amount of directory information to be maintained by a directory node that is close to the root can become very large, the directory information is usually organized as a cache to reduce its size, and maintains the inclusion property with respect to its children's caches/directories. This requires that on a replacement from a directory cache at a certain level of the tree, the replaced block be flushed out of all of its descendent directories in the tree as well. Similarly, replacement of the information about a block allocated within that subtree requires that copies of the block in nodes outside the subtree be invalidated or flushed.

A read miss from a node flows up the hierarchy until either (i) a directory indicates that its subtree has a copy (clean or dirty) of the memory block being requested, or (ii) the request reaches the directory that is the first common ancestor of the requesting node and the home node for that block, and that directory indicates that the block is not dirty outside that subtree. The request then flows down the hierarchy to the appropriate processing node to pick up the data. The data reply follows the same path back, updating the directories on its way. If the block was dirty, a copy of the block also finds its way to the home node.

A write miss in the cache flows up the hierarchy until it reaches a directory whose subtree contains the current owner of the requested memory block. The owner is either the home node, if the block is clean, or a dirty cache. The request travels down to the owner to pick up the data, and the requesting node becomes the new owner. If the block was previously in clean state, invalidations are propagated through the hierarchy to all nodes caching that memory block. Finally, all directories involved in the above transaction are updated to reflect the new owner and the invalidated copies.

In hierarchical snoopy schemes, the interconnection network is physically hierarchical to permit the snooping. With point to point communication, hierarchical directories do not need to rely on physically hierarchical interconnects. The hierarchy discussed above is a logical hierarchy, or a hierarchical data structure. It can be implemented on a network that is physically hierarchical (that is, an actual tree network with directory caches at the internal nodes and processing nodes at the leaves) or on a non-hierarchical network such as a mesh with the hierarchical directory embedded in this general network. In fact, there is a separate hierarchical directory structure for every block that is cached. Thus, the same physical node in a general network can be a leaf (processing) node for some blocks and an internal (directory) node for others (see Figure 8-41 below).

Finally, the storage overhead of the hierarchical directory has attractive scaling properties. It is the cost of the directory caches at each level. The number of entries in the directory goes up as we go further up the hierarchy toward to the root (to maintain inclusion without excessive replacements), but the number of directories becomes smaller. As a result, the total directory memory needed for all directories at any given level of the hierarchy is typically about the same. The directory storage needed is not proportional to the size of main memory, but rather to that of the caches in the processing nodes, which is attractive. The overall directory memory overhead is proportional to $\dfrac{C \times \log_b P}{M \times B}$, where C is the cache size per processing node at the leaf, M is the main-memory per node, B is the memory block size in bits, b is the branching factor of the hierarchy, and P is the number of processing nodes at the leaves (so $\log_b P$ is the number of levels in the tree). More information about hierarchical directory schemes can be found in the literature [Sco91, Wal92, Hag92, Joe95].

Performance Implications of Hierarchical Coherence

Hierarchical protocols, whether snoopy or directory, have some potential performance advantages. One is the combining of requests for a block as they go up and down the hierarchy. If a processing node is waiting for a memory block to arrive from the home node, another processing node that requests the same block can observe at their common ancestor directory that the block has already been requested. It can then wait at the intermediate directory and grab the block when it comes back, rather than send a duplicate request to the home node. This combining of transactions can reduce traffic and hence contention. The sending of invalidations and gathering of invalidation acknowledgments can also be done hierarchically through the tree structure. The other advantage is that upon a miss, if a nearby node in the hierarchy has a cached copy of the block, then the block can be obtained from that nearby node (cache to cache sharing) rather than having to go to the home which may be much further away in the network topology. This can reduce transit latency as well as contention at the home. Of course, this second advantage depends on how well locality in the hierarchy maps to locality in the underlying physical network, as well as how well the sharing patterns of the application match the hierarchy.

While locality in the tree network can reduce transit latency, particularly for very large machines, the overall latency and bandwidth characteristics are usually not advantageous for hierarchical schemes. Consider hierarchical snoopy schemes first. With busses there is a bus transaction and snooping latency at every bus along the way. With rings, traversing rings at every level of the hierarchy further increases latency to potentially very high levels. For example, the uncontended latency to access a location on a remote ring in a fully populated Kendall Square Research KSR1 machine [FBR93] was higher than 25 microseconds [SGC93], and a COMA organization was

used to reduce ring remote capacity misses. The commercial systems that have used hierarchical snooping have tended to use quite shallow hierarchies (the largest KSR machine was a two-level ring hierarchy, with up to 32 processors per ring). The fact that there are several processors per node also implies that the bandwidth between a node and its parent or child must be large enough to sustain their combined demands. The processors within a node will compete not only for bus or link bandwidth but also for the occupancy, buffers, and request tracking mechanisms of the node-to-network interface. To alleviate link bandwidth limitations near the root of the hierarchy, multiple buses or rings can be used closer to the root; however, bandwidth scalability in practical hierarchical systems remains quite limited.

For hierarchical directories, the latency problem is that the number of network transactions up and down the hierarchy to satisfy a request tends to be larger than in a flat memory-based scheme. Even though these transactions may be more localized, each one is a full-fledged network transaction which also requires either looking up or modifying the directory at its (intermediate) destination node. This increased end-point overhead at the nodes along the critical path tends to far outweigh any reduction in total transit latency on the network links themselves, especially given the characteristics of modern networks. Although some pipelining can be used—for example the data reply can be forwarded on toward the requesting node while a directory node is being updated—in practice the latencies can still become quite large compared to machines with no hierarchy [Hag92, Joe95]. Using a large branching factor in the hierarchy can alleviate the latency problem, but it increases contention. The root of the directory hierarchy can also become a bandwidth bottleneck, for both link bandwidth and directory lookup bandwidth. Multiple links may be used closer to the root—particularly appropriate for physically hierarchical networks [LAD+92]—and the directory cache may be interleaved among them. Alternatively, a multi-rooted directory hierarchy may be embedded in a non-hierarchical, scalable point-to-point interconnect [Wal92, Sco91, ScG93]. Figure 8-41 shows a possible organization. Like hierarchical
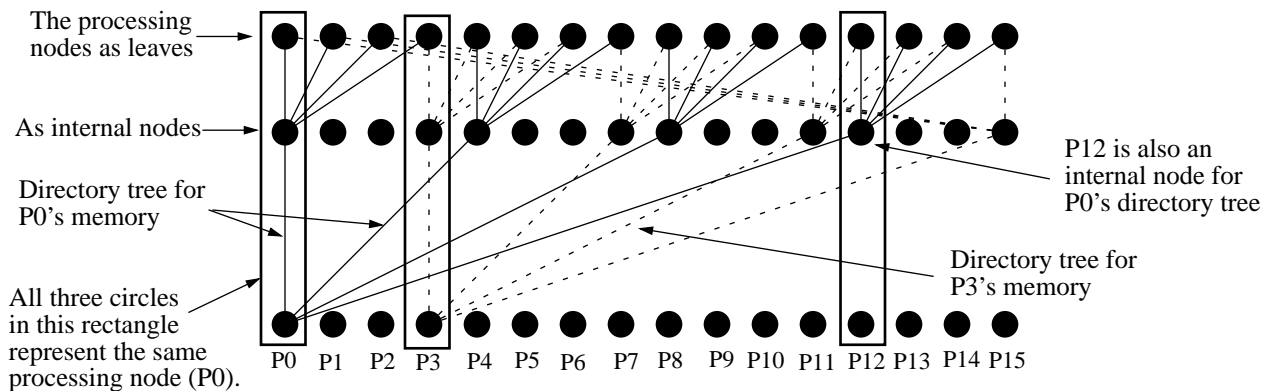
Figure 8-41 A multi-rooted hierarchical directory embedded in an arbitrary network.

A 16-node hierarchy is shown. For the blocks in the portion of main memory that is located at a processing node, that node itself is the root of the (logical) directory tree. Thus for P processing nodes, there are P directory trees. The figure shows only two of these. In addition to being the root for its local memory's directory tree, a processing node is also an internal node in the directory trees for the other processing nodes. The address of a memory block implicitly specifies a particular directory tree, and guides the physical traversals to get from parents to children and vice versa in this directory tree.

directory schemes themselves, however, these techniques have only been in the realm of research so far.

8.12 Concluding Remarks

Scalable systems that support a coherent shared address space are an important part of the multi-processing landscape. Hardware support for cache coherence is becoming increasingly popular for both technical and commercial workloads. Most of these systems use directory-based protocols, whether memory-based or cache-based. They are found to perform well, at least at the moderate scales at which they have been built, and to afford significant ease of programming compared to explicit message passing for many applications.

While supporting cache coherence in hardware has a significant design cost, it is alleviated by increasing experience, the appearance of standards, and the fact that microprocessors themselves provide support for cache coherence. With the microprocessor coherence protocol understood, the coherence architecture can be developed even before the microprocessor is ready, so there is not so much of a lag between the two. Commercial multiprocessors today typically use the latest microprocessors available at the time they ship.

Some interesting open questions for coherent shared address space systems include whether their performance on real applications will indeed scale to large processor counts (and whether significant changes to current protocols will be needed for this), whether the appropriate node for a scalable system will be a small scale multiprocessor or a uniprocessor, the extent to which commodity communication architectures will be successful in supporting this abstraction efficiently, and the success with which a communication assist can be designed that supports the most appropriate mechanisms for both cache coherence and explicit message passing.

8.13 References

[ABC+95] Anant Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of 22nd International Symposium on Computer Architecture*, pp. 2-13, June 1995.

[ALK+91] Anant Agarwal, Ben-Hong Lim, David Kranz, John Kubiatowicz. LimitLESS Directories. A Scalable Cache Coherence Scheme. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224-234, April 1991.

[ASH+88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of 15th International Symposium on Computer Architecture*, pp. 280-289, June 1988.

[CeF78] L. Censier and P. Feautrier. A New Solution to Cache Coherence Problems in Multiprocessor Systems. *IEEE Transaction on Computer Systems*, C-27(12):1112-1118, December 1978.

[ClA96] R. Clark and K. Alnes. An SCI Chipset and Adapter. In *Symposium Record, Hot Interconnects IV*, pp. 221-235, August 1996. Further information available from Data General Corporation.

[Con93] Convex Computer Corporation. Exemplar Architecture. Convex Computer Corporation, Richardson, Texas. November 1993.

[ENS+95] Andrew Erlichson, Basem Nayfeh, Jaswinder Pal Singh and Oyekunle Olukotun. The Benefits of Clustering in Cache-coherent Multiprocessors: An Application-driven Investigation. In Proceedings of Supercomputing'95, November 1995.

[FBR93] Steve Frank, Henry Burkhardt III, James Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of COMPCON*, pp. 285-294, Spring 1993.

[FCS+93]   Jean-Marc Frailong, et al. The Next Generation SPARC Multiprocessing System Architecture. In *Proceedings of COMPCON*, pp. 475-480, Spring 1993.

[FVS92]   K. Farkas, Z. Vranesic, and M. Stumm. Cache Consistency in Hierarchical Ring-Based Multiprocessors. In *Proceedings of Supercomputing '92*, November 1992.

[HHS+95]   Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg and John L. Hennessy. The Effects of Latency, Occupancy and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report no. CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[Gus92]   David Gustavson. The Scalable Coherence Interface and Related Standards Projects. *IEEE Micro*, 12(1), pp:10-22, February 1992.

[GuW92]   Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794-810, July 1992.

[GWM90]   Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache-Coherence Schemes. In *Proceedings of International Conference on Parallel Processing,* volume I, pp. 312-321, August 1990.

[Hag92]   Erik Hagersten. Toward Scalable Cache Only Memory Architectures. Ph.D. Dissertation, Swedish Institute of Computer Science, Sweden, October 1992.

[HLK97]   Cristina Hristea, Daniel Lenoski and John Keen. Measuring Memory Hierarchy Performance of Cache Coherent Multiprocessors Using Micro Benchmarks. In *Proceedings of SC97*, 1997.

[IEE93]   IEEE Computer Society. IEEE Standard for Scalable Coherent Interface (SCI). IEEE Standard 1596-1992, New York, August 1993.

[IEE95]   IEEE Standard for Cache Optimization for Large Numbers of Processors using the Scalable Coherent Interface (SCI) Draft 0.35, September 1995.

[Joe95]   Truman Joe. COMA-F A Non-Hierarchical Cache Only Memory Architecture. Ph.D. thesis, Computer Systems Laboratory, Stanford University, March 1995.

[Kax96]   Stefanos Kaxiras. Kiloprocessor extensions to SCI. In Proceedings of the Tenth International Parallel Processing Symposium, 1996.

[KaG96]   Stefanos Kaxiras and James Goodman. The GLOW Cache Coherence Protocol Extensions for Widely Shared Data. In Proceedings of the International Conference on Supercomputing, pp. 35-43, 1996.

[KOH+94]   Jeff Kuskin, Dave Ofelt, Mark Heinrich, John Heinlein, Rich Simoni, Kourosh Gharachorloo, John Chapin, Dave Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 302-313, April 1994.

[LRV96]   James R. Larus, Brad Richards and Guhan Viswanathan. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In Gregory V. Wilson and Paul Lu, eds., Parallel Programming Using C++, MIT Press 1996.

[Lal97]   James P. Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[LAD+92]   C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, R. Zak. The Network Architecture of the CM-5. Symposium on Parallel and Distributed Algorithms '92" Jun 1992, pp. 272-285.

[Len92]   Daniel Lenoski. The Stanford DASH Multiprocessor... PhD Thesis, Stanford University, 1992 <<xxx?>>.

[LLG+90]   Daniel Lenoski, et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 148-159, May 1990.

[LLJ+93]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[LoC96]   Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of 23rd International Symposium on Computer Architecture*, pp. 308-317, 1996.

[OKN90]   B. O'Krafka and A. Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 138-147, May 1990.

[RSG93]   Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 14-25, May 1993. <unused>

[SGC93]   Rafael Saavedra, R. Stockton Gaines, and Michael Carlton. Micro Benchmark Analysis of the KSR1. In *Proceedings of Supercomputing '93*, pp. 202-213, November 1993.

[Sco91]   Steven Scott. A Cache-Coherence Mechanism for Scalable Shared-Memory Multiprocessors. In *Proceedings of International Symposium on Shared Memory Multiprocessing*, pages 49-59, April 1991.

[ScG93]   Steven Scott and James Goodman. Performance of Pruning Cache Directories for Large-Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):520-534, May 1993.

[SiH91]   Rich Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proceedings of International Symposium on Shared Memory Multiprocessing*, pages 72-81, April 1991.

[SFG+93]   Pradeep Sindhu, et al. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. In *Proceedings of COMPCON*, pp. 338-344, Spring 1993.

[Sin97]   Jaswinder Pal Singh. Some Aspects of the Implementation of the SGI Origin2000 Hub Controller. Technical Report no. xxx, Computer Science Department, Princeton University. Also on home page of *Parallel Computer Architecture* by David Culler and Jaswinder Pal Singh at Morgan Kaufman Publishers.

[Tan76]   C. Tang. Cache Design in a Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings*, pp. 749-753, June 1976.

[ThD90]   Manu Thapar and Bruce Delagi. Stanford Distributed-Directory Protocol. In *IEEE Computer*, 23(6):78-80, June 1990.

[THH+96]   Radhika Thekkath, Amit Pal Singh, Jaswinder Pal Singh, John Hennessy and Susan John. An Application-Driven Evaluation of the Convex Exemplar SP-1200. *In Proceedings of the International Parallel Processing Symposium*, June 1997. (unused)

[Tuc86]   S. Tucker. The IBM 3090 System: An Overview. *IBM Systems Journal*, 25(1):4-19, January 1986.

[VSL+91]   Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A Hierarchically Structured Shared Memory Multiprocessor. *IEEE Computer*, 24(1), pp. 72-78, January 1991.

[Wal92]   Deborah A. Wallach. PHD: A Hierarchical Cache Coherence Protocol. S.M. Thesis. Massachusetts Institute of Technology, 1992. Available as Technical Report No. 1389, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August 1992.

[Web93]   Wolf-Dietrich Weber. Scalable Directories for Cache-coherent Shared-memory Multiprocessors. Ph.D. Thesis, Computer Systems Laboratory, Stanford University, January 1993. Also available as Stanford University Technical Report no. CSL-TR-93-557.

[WGH+97]   Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki and Winifrid Wilcke. The Mercury Interconnect Architecture: A Cost-effective Infrastructure for High-performance Servers. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[WeS94]   Shlomo Weiss and James Smith. Power and PowerPC. Morgan Kaufmann Publishers Inc. 1994. <unused>

[WiC80]   L. Widdoes, Jr., and S. Correll. The S-1 Project: Developing High Performance Computers. In *Proceedings of COMPCON*, pp. 282-291, Spring 1980.

[Wil87]   Andrew Wilson Jr. Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 244-252, June 1987.

[WWS+89]   P. Woodbury, A. Wilson, B. Shein, I. Gertner, P.Y. Chen, J. Bartlett, and Z. Aral. Shared Memory Multiprocessors: The Right Approach to Parallel Processing. In *Proceedings of COMPCON*, pp. 72-80, Spring 1989.

8.14 Exercises

8.1 Short answer questions

    a. What are the inefficiencies and efficiencies in emulating message passing on a cache coherent machine compared to the kinds of machines discussed in the previous chapter?

    b. For which of the case study parallel applications used in this book do you expect there to be a substantial advantage to using multiprocessor rather than uniprocessor nodes (assuming the same total number of processors). For which do you think there might be disadvantages in some machines, and under what circumstances? Are there any special benefits that the Illinois coherence scheme offers for organizations with multiprocessor nodes?

    c. How might your answer to the previous question differ with increasing scale of the machine. That is, how do you expect the performance benefits of using fixed-size multiprocessor nodes to change as the machine size is increased to hundreds of processors?

8.2 High-level memory-based protocol tradeoffs.

    a. Given a 512 processor system, in which each node visible to the directory is has 8 processors and 1GB of main memory, and a cache block size of 64 bytes, what is the directory memory overhead for:

    (i) a Full Bit Vector Scheme

    (ii) DIR_iB, with i = 3.

    b. In a full bit vector scheme, why is the volume of data traffic guaranteed to be much greater than that of coherence traffic (i.e. invalidation and invalidation-acknowledgment messages)?

    c. The chapter provided diagrams showing the network transactions for strict request-reply, intervention forwarding and reply forwarding for read operations in a flat memory-based protocol like that of the SGI Origin. Do the same for write operations.

d. The Origin protocol assumed that acknowledgments for invalidations are gathered at the requestor. An alternative is to have the acknowledgments be sent back to the home (from where the invalidation requests come) and have the home send a single acknowledgment back to the requestor. This solution is used in the Stanford FLASH multiprocessor. What are the performance and complexity tradeoffs between these two choices?

e. Under what conditions are replacement hints useful and under what conditions are they harmful? In which of our parallel applications do you think they would be useful or harmful to overall performance, and why?

f. Draw the network transaction diagrams (like those in Figure 8-16 on page 549) for an uncached read shared request, and uncached read exclusive request, and a write-invalidate request in the Origin protocol. State one example of a use of each.

8.3 High-level issues in cache-based protocols.

a. Instead of the doubly linked list used in the SCI protocol, it is possible to use a single linked list. What is the advantage? Describe what modifications would need to be made to the following operations if a singly linked list were used:

(i) replacement of a cache block that is in a sharing list

(ii) write to a cache block that is in a sharing list.

Qualitatively discuss the effects this might have on large scale multiprocessor performance?

b. How might you reduce the latency of writes that cause invalidations in the SCI protocol? Draw the network transactions. What are the major tradeoffs?

8.4 Adaptive protocols.

a. When a variable exhibits migratory sharing, a processor that reads the variable will be the next one to write it. What kinds of protocol optimizations could you use to reduce traffic and latency in this case, and how would you detect the situation dynamically. Describe a scheme or two in some detail.

b. Another pattern that might be detected dynamically is a producer-consumer pattern, in which one processor repeatedly writes (produces) a variable and another processor repeatedly reads it. Is the standard MESI invalidation-based protocol well-suited to this? Why or why not? What enhancements or protocol might be better, and what are the savings in latency or traffic? How would you dynamically detect and employ the enhancements or variations?

8.5 Correctness issues in protocols

a. Why is write atomicity more difficult to provide with update protocols in directory-based systems? How would you solve the problem? Does the same difficulty exist in a bus-based system?

b. Consider the following program fragment running on a cache-coherent multiprocessor, assuming all values to be 0 initially.

P1	P2	P3	P4
A = 1	u = A	w = A	A = 2
	v = A	x = A	

There is only one shared variable (A). Suppose that a writer magically knows where the cached copies are, and sends updates to them directly without consulting a directory node.

Construct a situation in which write atomicity may be violated, assuming an update-based protocol.

(i) Show the violation of SC that occurs in the results.

(ii) Can you produce a case where coherence is violated as well? How would you solve these problems?

(iii) Can you construct the same problems for an invalidation-based protocol?

(iv) Could you do it for update protocols on a bus?

c. In handling writebacks in the Origin protocol, we said that when the node doing the write-back receives an intervention it ignores it. Given a network that does not preserve point to point order, what situations do we have to be careful of? How do we detect that this intervention should be dropped? Would there be a problem with a network that preserved point-to-point order?

d. Can the serialization problems discussed for Origin in this chapter arise even with a strict request-reply protocol, and do the same guidelines apply? Show example situations, including the examples discussed in that section.

e. Consider the serialization of writes in NUMA-Q, given the two-level hierarchical coherence protocol. If a node has the block dirty in its remote cache, how might writes from other nodes that come to it get serialized with respect to writes from processors in this node. What transactions would have to be generated to ensure the serialization?

8.6 Eager versus delayed replies to processor.

a. In the Origin protocol, with invalidation acknowledgments coming to the home rather than the requestor, it would be possible for other read and write requests to come to the requestor that has outstanding invalidations before the acknowledgments for those invalidations come in.

(i) Is this possible or a problem with delayed exclusive replies? With eager exclusive replies? If it is a problem, what is the simplest way to solve it?

(ii) Suppose you did indeed want to allow the requestor with invalidations outstanding to process incoming read and write requests. Consider write requests first, and construct an example where this can lead to problems. (Hint: consider the case where P1 writes to a location A, P2 writes to location B which is on the same cache block as A and then writes a flag, and P3 spins on the flag and then reads the value of location B.) How would you allow the incoming write to be handled but still maintain correctness? Is it easier if invalidation acknowledgments are collected at the home or at the requestor?

(iii) Now answer the same questions above for incoming read requests.

(iv) What if you were using an update-based protocol. Now what complexities arise in allowing an incoming request to be processed for a block that has updates outstanding from a previous write, and how might you solve them?

(v) Overall, would you choose to allow incoming requests to be processed while invalidations or updates are outstanding, or deny them?

b. Suppose a block needs to be written back while there are invalidations pending for it. Can this lead to problems, or is it safe? If it is problematic, how might you address the problem?

c. Are eager exclusive replies useful with an underlying SC model? Are they at all useful if the processor itself provides out of order completion of memory operations, unlike the MIPS R10000?

    d. Do the tradeoffs between collecting acknowledgments at the home and at the requestor change if eager exclusive replies are used instead of delayed exclusive replies?

8.7 Implementation Issues.

    e. In the Origin implementation, incoming request messages to the memory/directory interface are given priority over incoming replies, unless there is a danger of replies being starved. Why do you think this choice of giving priorities to requests was made? Describe some alternatives for how you might you detect when to invert the priority. What would be the danger with replies being starved?

8.8 4. Page replication and migration.

    a. Why is it necessary to flush TLBs when doing migration or replication of pages?

    b. For a CC-NUMA multiprocessor with software reloaded TLBs, suppose a page needs to be migrated. Which one of the following TLB flushing schemes would you pick and why? (Hint: The selection should be based on the following two criteria: Cost of doing the actual TLB flush, and difficulty of tracking necessary information to implement the scheme.)

    (i) Only TLBs that currently have an entry for a page

    (ii) Only TLBs that have loaded an entry for a page since the last flush

    (iii) All TLBs in the system.

    c. For a simple two processor CC-NUMA system, the traces of cache misses for 3 virtual pages X, Y, Z from the two processors P0 and P1 are shown. Time goes from left to right. "R" is a read miss and "W" is a write miss. There are two memories M0 and M1, local to P0 and P1 respectively. A local miss costs 1 unit and a remote miss costs 4 units. Assume that read misses and write misses cost the same.

    Page X:

        P0: RRRR R R RRRRR   RRR

        P1: R R R R R   RRRR RR

    Page Y:

        P0: no accesses

        P1: RR WW RRRR RWRWRW WWWR

    Page Z:

        P0: R W RW R R RRWRWRWRW

        P1: WR RW RW W W   R

    (i) How would you place pages X, Y, and Z, assuming complete knowledge of the entire trace?

    (ii) Assume the pages were initially placed in M0, and that a migration or replication can be done at zero cost. You have prior knowledge of the entire trace. You can do one migration, or one replication, or nothing for each page at the beginning of the trace. What action would be appropriate for each of the pages?

    (iii) Same as (ii) , but a migration or replication costs 10 units. Also give the final memory access cost for each page.

    (iv) Same as (iii), but a migration or replication costs 60 units.

    (v) Same as (iv), but the cache-miss trace for each page is the shown trace repeated 10 times. (You still can only do one migration or replication at the beginning of the entire trace)

8.9 Synchronization:

a. Full-empty bits, introduced in Section 5.6, provide hardware support for fine-grained synchronization. What are the advantages and disadvantages of full-empty bits, and why do you think they are not used in modern systems?

b. With an invalidation based protocol, lock transfers take more network transactions than necessary. An alternative to cached locks is to use uncached locks, where the lock variable stays in main memory and is always accessed directly there.

(i) Write pseudocode for a simple lock and a ticket lock using uncached operations?

(ii) What are the advantages and disadvantages relative to using cached locks? Which would you deploy in a production system?

c. Since high-contention and low-contention situations are best served by different lock algorithms, one strategy that has been proposed is to have a library of synchronization algorithms and provide hardware support to switch between them "reactively" at runtime based on observed access patterns to the synchronization variable [LiA9??].

(i) Which locks would you provide in your library?

(ii) Assuming a memory-based directory protocol, design simple hardware support and a policy for switching between locks at runtime.

(iii) Describe an example where this support might be particularly useful.

(iv) What are the potential disadvantages?

8.10 Software Implications and Performance Issues

a. You are performing an architectural study using four applications: Ocean, LU, an FFT that performs local calculations on rows separated by a matrix transposition, and Barnes-Hut. For each application, answer the above questions assuming a CC-NUMA system:

(i) what modifications or enhancements in data structuring or layout would you use to ensure good interactions with the extended memory hierarchy?

(ii) what are the interactions with cache size and granularities of allocation, coherence and communication that you would be particularly careful to represent or not represent?

b. Consider the example of transposing a matrix of data in parallel, as is used in computations such as high-performance Fast Fourier Transforms. Figure 8-42 shows the transpose pictorially. Every processor tranposes one "patch" of its assigned rows to every other processor, including one to itself. Before the transpose, a processor has read and written its assigned rows of the source matrix of the transpose, and after the transpose it reads and writes its assigned rows of the destination matrix. The rows assigned to a processor in both the source and destination matrix are allocated in its local memory. There are two ways to perform the transpose: a processor can read the local elements from its rows of the source matrix and write them to the associated elements of the destination matrix, whether they are local or remote, as shown in the figure; or a processor can write the local rows of the destination matrix and read the associated elements of the source matrix whether they are local or remote.

(i) Given an invalidation-based directory protocol, which method do you think will perform better and why?

(iii) How do you expect the answer to (i) to change if you assume an update-based directory protocol?
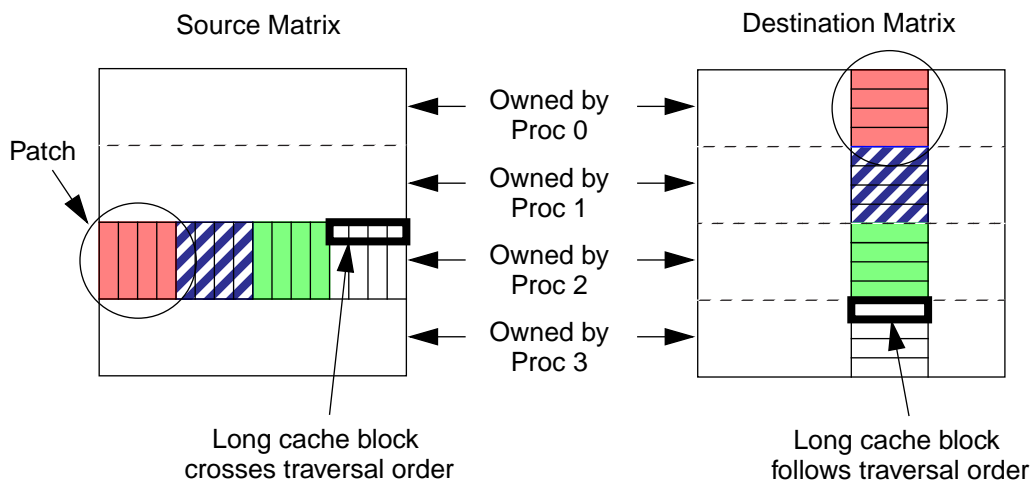
Figure 8-42 Sender-initiated matrix transposition.

The source and destination matrices are partitioning among processes in groups of contiguous rows. Each process divides its set of n/p rows into p patches of size n/p * n/p. Consider process P2 as a representative example: One patch assigned to it ends up in the assinged set of rows of every other process, and it transposes one patch (third from left, in this case) locally.

Consider this implementation of a matrix transpose which you plan to run on 8 processors. Each processor has one level of cache, which is fully associative, 8 KB, with 128 byte lines. (*Note: AT and A are not the same matrix*):

```
Transpose(double **A, double **AT)
{
    int i,j,mynum;

    GETPID(mynum);

    for (i=mynum*nrows/p; i<((mynum+1)*(nrows/p)); i++) {
        for (j=0; j<1024; j++) {
            AT[i][j] = A[j][i];
        }
    }
}
```

The input data set is a 1024-by-1024 matrix of double precision floating point numbers, decomposed so that each processor is responsible for generating a contiguous block of rows in the transposed matrix AT.

Ignoring the contention problem caused by all processors first going to processor 0, what is the major performance problem with this code? What technique would you use to solve it?

Restructure the code to alleviate the performance problem as much as possible. Write the entire restructured loop.

8.11 Bus Hierarchies.

a. Consider a hierarchical bus-based system with a centralized memory at the root of the hierarhcy rather than distributed memory. What would be the main differences in how reads and writes are satisfied? Briefly describe in terms of the path taken by reads and writes.

b. Could you have a hierarchical bus based system with centralized memory (say) without pursuing the inclusion property between the L2 (node) cache and the L1 caches? If so, what complications would it cause?

c. Given the hierarchical bus-based multiprocessor shown in Figure 8-37, design a state-transition diagram for maintaining coherence for the L2 cache. Assume that the L1 caches follow the Illinois protocol. Precisely define the cache block states for L2, the available bus-transaction types for the B1 and B2 buses, other actions possible, and the state-transition diagram. Note you have considerable freedom in choosing cache block states and other parameters; there is no one fixed right answer. Justify your decisions.

d. Given the ending state of caches in examples for Figure 8-37 on page 596, give some more memory references, and ask for detailed description of coherence actions that would need to occur.

e. [To ensure SC in a two-level hierarchical bus design, is it okay to return an acknowledgment when the invalidation request reaches the B2 bus? If yes, what constraints are imposed on the design/implementation of L1 caches and L2 caches and the orders preserved among transactions? If not, why not. Would it be okay if the hierarchy had more than two levels?

f. Suppose two processors in two different nodes of a hierarchical bus-based machine issue an upgrade for a block at the same time. Trace their paths through the system, discussing all state changes and when they must happen, as well as what precautions prevent deadlock and prevent both processors from gaining ownership.

g. An optimization in distributed-memory bus-based hierarchies is that if another processor's cache on the local bus can supply the data, we do not have to go to the global bus and remote node. What are the tradeoffs of supporting this optimization in ring-based hierarchies? (Answer: May need to go around the local ring an extra time to get to the inter-ring interface.)

8.12 Consider design of a single chip in year 2005, when you have 100 million transistors. You decide to put four processors on the chip. How would you design the interface to build a modular component that can be used in scalable systems: (i) global-memory organization (e.g., the chip has multiple CPUs each with local cache, a shared L2 cache, and a single snoopy cache bus interface), or (ii) distributed memory organization (e.g., the chip has multiple CPUs each with local cache on an internal snoopy bus, a separate memory bus interface that comes of this internal snoopy bus, and another separate snoopy cache bus interface). Discuss the tradeoffs.??? not very well formed???

8.13 Hierarchical directories.

a. What branching would you choose in a machine with a hierarchical directory? Highlight the major tradeoffs. What techniques might you use to alleviate the performance tradeoffs? Be as specific in your description as possible.

b. Is it possible to implement hierarchical directories without maintaining inclusion in the directory caches. Design a protocol that does that and discuss the advantages and disadvantages.

CHAPTER 9   # Hardware-Software Tradeoffs

9.1 Introduction

This chapter focuses on addressing some potential limitations of the directory-based, cache-coherent systems discussed in the previous chapter, and the hardware-software tradeoffs that this engenders. The primary limitations of those systems, in both performance and cost, are the following.

- *High waiting time at memory operations.* Sequential consistency (SC) was assumed to be the memory consistency model of choice, and the chapter discussed how systems might satisfy the sufficient conditions for SC. To do this, a processor would have to wait for the previous memory operation to complete before issuing the next one. This has even greater impact on performance in scalable systems than in bus-based systems, since communication latencies are longer and there are more network transactions in the critical path. Worse still, it is very limiting for compilers, which cannot reorder memory operations to shared data at all if the programmer assumes sequential consistency.

- *Limited capacity for replication*. Communicated data are automatically replicated only in the processor cache, not in local main memory. This can lead to capacity misses and artifactual communication when working sets are large and include nonlocal data or when there are a lot of conflict misses.

- *High design and implementation cost*. The communication assist contains hardware that is specialized for supporting cache-coherence and tightly integrated into the processing node. Protocols are also complex, and getting them right in hardware takes substantial design time. By cost here we mean the cost of hardware and of system design time. Recall from Chapter 3 that there is also a programming cost associated with achieving good performance, and approaches that reduce system cost can often increase this cost dramatically.

The chapter will focus on these three limitations, the first two of which have primarily to do with performance and the third primarily with cost. The approaches that have been developed to address them are still controversial to varying extents, but aspects of them are finding their way into mainstream parallel architecture. There are other limitations as well, including the address-ability limitations of a shared physical address space—as discussed for the Cray T3D in Chapter 7—and the fact that a single protocol is hard-wired into the machine. However, solutions to these are often incorporated in solutions to the others, and they will be discussed as advanced topics.

The problem of waiting too long at memory operations can be addressed in two ways in hardware. First, the implementation can be designed to not satisfy the sufficient conditions for SC, which modern non-blocking processors are not inclined to do anyway, but to satisfy the SC model itself. That is, a processor needn't wait for the previous operation to complete before issuing the next one; however, the system ensures that operations do not complete or become visible to other processors out of program order. Second, the memory consistency model can itself be relaxed so program orders do not have to be maintained so strictly. Relaxing the consistency model changes the semantics of the shared address space, and has implications for both hardware and software. It requires more care from the programmer in writing correct programs, but enables the hardware to overlap and reorder operations further. It also allows the compiler to reorder memory operations within a process before they are even presented to hardware, as optimizing compilers are wont to do. Relaxed memory consistency models will be discussed in Section 9.2.

The problem of limited capacity for replication can be addressed by automatically caching data in main memory as well, not just in the processor caches, and by keeping these data coherent. Unlike in hardware caches, replication and coherence in main memory can be performed at a variety of granularities—for example a cache block, a page, or a user-defined object—and can be managed either directly by hardware or through software. This provides a very rich space of protocols, implementations and cost-performance tradeoffs. An approach directed primarily at improving performance is to manage the local main memory as a hardware cache, providing replication and coherence at cache block granularity there as well. This approach is called cache-only memory architecture or COMA, and will be discussed in Section 9.3. It relieves software from worrying about capacity misses and initial distribution of data across main memories, while still providing coherence at fine granularity and hence avoiding false-sharing. However, it is hardware-intensive, and requires per-block tags to be maintained in main memory as well.

Finally, there are many approaches to reduce hardware cost. One approach is to integrate the communication assist and network less tightly into the processing node, increasing communication latency and occupancy. Another is to provide automatic replication and coherence in soft-

ware rather than hardware. The latter approach provides replication and coherence in main memory, and can operate at a variety of granularities. It enables the use of off-the-shelf commodity parts for the nodes and interconnect, reducing hardware cost but pushing much more of the burden of achieving good performance on to the programmer. These approaches to reduce hardware cost are discussed in Section 9.4.

The three issues are closely related. For example, cost is strongly related to the manner in which replication and coherence are managed in main memory: The coarser the granularities used, the more likely that the protocol can be synthesized through software layers, including the operating system, thus reducing hardware costs (see Figure 9-1). Cost and granularity are also related to the
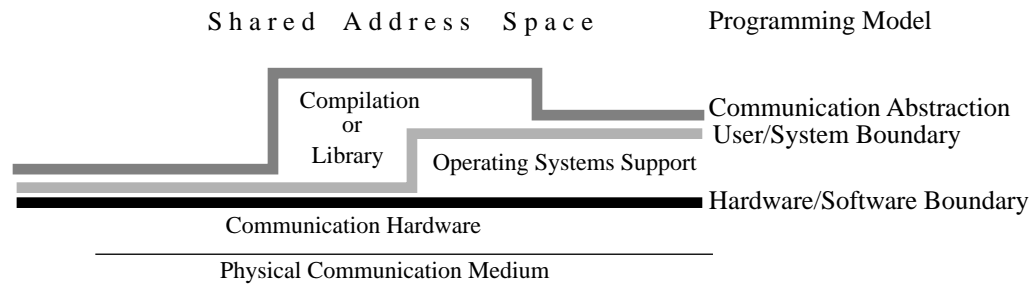
Figure 9-1 Layers of the communication architecture for systems discussed in this chapter.

memory consistency model: Lower-cost solutions and larger granularities benefit more from relaxing the memory consistency model, and implementing the protocol in software makes it easier to fully exploit the relaxation of the semantics. To summarize and relate the alternatives, Section 9.5 constructs a framework for the different types of systems based on the granularities at which they allocate data in the replication store, keep data coherent and communicate data. This leads naturally to an approach that strives to achieve a good compromise between the high-cost COMA approach and the low-cost all-software approach. This approach, called Simple-COMA, is discussed in Section 9.5 as well.

The implications of the systems discussed in this chapter for parallel software, beyond those discussed in the previous chapters, are explored in Section 9.6. Finally, Section 9.7 covers some advanced topics, including the techniques to address the limitations of a shared physical address space and a fixed coherence protocol.

9.2 Relaxed Memory Consistency Models

Recall from Chapter 5 that the memory consistency model for a shared address space specifies the constraints on the order in which memory operations (to the same or different locations) can appear to execute with respect to one another, enabling programmers to reason about the behavior and correctness of their programs. In fact, any system layer that supports a shared address space naming model has a memory consistency model: the programming model or programmer's interface, the communication abstraction or user-system interface, and the hardware-software interface. Software that interacts with that layer must be aware of its memory consistency model. We shall focus mostly on the consistency model as seen by the programmer—i.e. at the interface

between the programmer and the rest of the system composed of the compiler, OS and hardware—since that is the one with which programmers reason[1]. For example, a processor may preserve all program orders presented to it among memory operations, but if the compiler has already reordered operations then programmers can no longer reason with the simple model exported by the hardware.

Other than programmers (including system programmers), the consistency model at the programmer's interface has implications for programming languages, compilers and hardware as well. To the compiler and hardware, it indicates the constraints within which they can reorder accesses from a process and the orders that they cannot appear to violate, thus telling them what tricks they can play. Programming languages must provide mechanisms to introduce constraints if necessary, as we shall see. In general, the fewer the reorderings of memory accesses from a process we allow the system to perform, the more intuitive a programming model we provide to the programmer, but the more we constrain performance optimizations. The goal of a memory consistency model is to impose ordering constraints that strike a good balance between programming complexity and performance. The model should also be portable; that is, the specification should be implementable on many platforms, so that the same program can run on all these platforms and preserve the same semantics.

The sequential consistency model that we have assumed so far provides an intuitive semantics to the programmer—program order within each process and a consistent interleaving across processes—and can be implemented by satisfying its sufficient conditions. However, its drawback is that it by preserving a strict order among accesses it restricts many of the performance optimizations that modern uniprocessor compilers and microprocessors employ. With the high cost of memory access, computer systems achieve higher performance by reordering or overlapping the servicing of multiple memory or communication operations from a processor. Preserving the sufficient conditions for SC clearly does not allow for much reordering or overlap. With SC at the programmer's interface, the compiler cannot reorder memory accesses even if they are to different locations, thus disallowing critical performance optimizations such as code motion, common-subexpression elimination, software pipelining and even register allocation.

Example 9-1   Show how register allocation can lead to a violation of SC even if the hardware satisfies SC.

Answer   Consider the code fragment shown in Figure 9-2(a). After register allocation, the code produced by the compiler and seen by hardware might look like that in Figure 9-2(b). The result (u,v) = (0,0) is disallowed under SC hardware in (a), but not only can but will be produced by SC hardware in (b). In effect, register allocation reorders the write of A and the read of B on P1 and reorders the write of B and the read of A on P2. A uniprocessor compiler might easily perform these

---

1. The term programmer here refers to the entity that is responsible for generating the parallel program. If the human programmer writes a sequential program that is automatically parallelized by system software, then it is the system software that has to deal with the memory consistency model; the programmer simply assumes sequential semantics as on a uniprocessor.

optimizations in each process: They are valid for sequential programs since the reordered accesses are to different locations.

P1	P2
B = 0	A = 0
A = 1	B = 1
u = B	v = A

P1	P2
r1 = 0	r2 = 0
A = 1	B = 1
u = r1	v = r2
B = r1	A = r2

(a) Before register allocation

(a) After register allocation

Figure 9-2 Example showing how register allocation by the compiler can violate SC.

The code in (a) is the original code with which the programmer reasons under SC. r1, r2 are registers.

SC at the programmer's interface implies SC at the hardware-software interface; if the sufficient conditions for SC are met, a processor waits for an access to complete or at least commit before issuing the next one, so most of the latency suffered by memory references is directly seen by processors as stall time. While a processor may continue executing non-memory instructions while a single outstanding memory reference is being serviced, the expected benefit from such overlap is tiny (even without instruction-level parallelism on average every third instruction is a memory reference [HeP95]). We need to do something about this performance problem.

One approach we can take is to preserve sequential consistency at the programmer's interface but hide the long latency stalls from the processor. We can do this in several ways, divided into two categories [GGH91]. The techniques and their performance implications will be discussed further in Chapter 11; here, we simply provide a flavor for them. In the first category, the system still preserves the sufficient conditions for SC. The compiler does not reorder memory operations. Latency tolerance techniques such as prefetching of data or multithreading (see Chapter 11) are used to overlap accesses with one another or with computation—thus hiding much of their latency from the processor—but the actual read and write operations are not issued before previous ones complete and the operations do not become visible out of program order.

In the second category, the system preserves SC but not the sufficient conditions at the programmer's interface. The compiler can reorder operations as long as it can guarantee that sequential consistency will not be violated in the results. Compiler algorithms have been developed for this [ShS88,KrY94], but they are expensive and their analysis is currently quite conservative. At the hardware level, memory operations are issued and executed out of program order, but are guaranteed to become visible to other processors in program order. This approach is well suited to dynamically scheduled processors that use an instruction lookahead buffer to find independent instructions to issue. The instructions are inserted in the lookahead buffer in program order, they are chosen from the instruction lookahead buffer and executed out of order, but they are guaranteed to retire from the lookahead buffer in program order. Operations may even issue and execute out of order past an unresolved branch in the lookahead buffer based on branch prediction— called speculative execution—but since the branch will be resolved and retire before them they will not become visible to the register file or external memory system before the branch is resolved. If the branch was mis-predicted, the effects of those operations will never become visi-

ble. The technique called *speculative reads* goes a little further; here, the values returned by reads are used even before they are known to be correct; later checks determine if they were incorrect, and if so the computation is rolled back to reissue the read. Note that it is not possible to speculate with stores in this manner because once a store is made visible to other processors it is extremely difficult to roll back and recover: a store's value should not be made visible to other processors or the external memory system environment until all previous references have correctly completed.

Some or all of these techniques are supported by many modern microprocessors, such as the MIPS R10000, the HP PA-8000, and the Intel Pentium Pro. However, they do require substantial hardware resources and complexity, their success at hiding multiprocessor latencies is not yet clear (see Chapter 11), and not all processors support them. Also, these techniques work for processors, but they do not help compilers perform the reorderings of memory operations that are critical for their optimizations.

A completely different way to overcome the performance limitations imposed by SC is to change the memory consistency model itself; that is, to not guarantee such strong ordering constraints to the programmer, but still retain semantics that are intuitive enough to a programmer. By relaxing the ordering constraints, these *relaxed consistency models* allow the compiler to reorder accesses before presenting them to the hardware, at least to some extent. At the hardware level, they allow multiple memory accesses from the same process to be outstanding at a time and even to complete or become visible out of order, thus allowing much of the latency to be overlapped and hidden from the processor. The intuition behind relaxed models is that SC is usually too conservative, in that many of the orders it preserves are not really needed to satisfy a programmer's intuition in most situations. Much more detailed treatments of relaxed memory consistency models can be found in [Adv93,Gha95].

Consider the simple example shown in Figure 9-3. On the left are the orderings that will be main-

P1   P2   P1   P2

A = 1;   while (flag == 0);   A = 1;   ... = flag;

B = 1;   u = A;   B = 1;   ... = flag;

flag = 1;   v = B;   flag = 1;   ... = flag;

                                                                              u = A;

                                                                              v = B;

Orderings Maintained By   **Orderings Necessary For**
Sequential Consistency   **Correct Program Semantics**
(a)   (b)
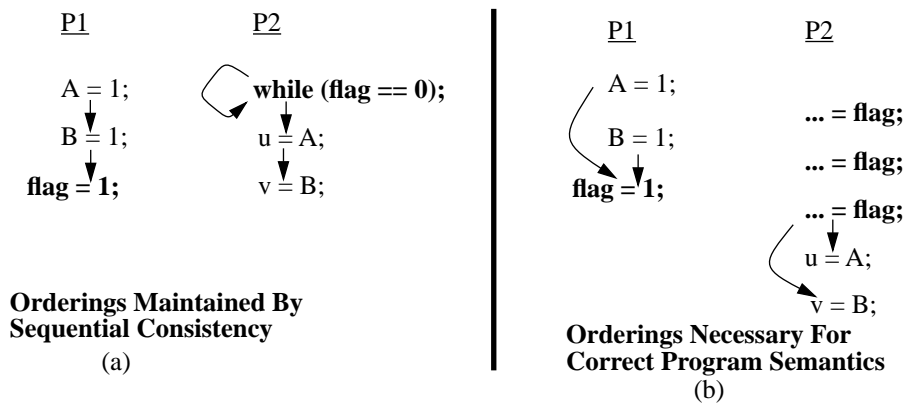
Figure 9-3 Intuition behind relaxed memory consistency models.

The arrows in the figure indicate the orderings maintained. The left side of the figure (part (a)) shows the orderings maintained by the sequential consistency model. The right side of the figure (part (b)) shows the orderings that are *necessary* for "correct / intuitive" semantics.

tained by an SC implementation. On the right are the orderings that are necessary for intuitively correct program semantics. The latter are far fewer. For example, writes to variables A and B by P1 can be reordered without affecting the results observed by the program; all we must ensure is

that both of them complete before variable `flag` is set to 1. Similarly, reads to variables A and B can be reordered at P2, once `flag` has been observed to change to value 1.[1] Even with these reorderings, the results look just like those of an SC execution. On the other hand, while the accesses to `flag` are also simple variable accesses, a model that allowed them to be reordered with respect to A and B at either process would compromise the intuitive semantics and SC results. It would be wonderful if system software or hardware could automatically detect which program orders are critical to maintaining SC semantics, and allow the others to be violated for higher performance [ShS88]. However, the problem is undecidable for general programs, and inexact solutions are often too conservative to be very useful.

There are three parts to a complete solution for a relaxed consistency model: the system specification, the programmer's interface, and a translation mechanism.

> **The system specification**. This is a clear specification of two things. First, what program orders among memory operations are guaranteed to be preserved, in an observable sense, by the system, including whether write atomicity will be maintained. And second, if not all program orders are preserved by default, then what mechanisms does the system provide for a programmer to enforce those orderings explicitly. As should be clear by now, each of the compiler and the hardware has a system specification, but we focus on the specification that the two together or the system as a whole presents to the programmer. For a processor architecture, the specification it exports governs the reorderings that it allows and the order-preserving primitives it provides, and is often called the processor's memory model.

> **The programmer's interface**. The system specification is itself a consistency model. A programmer may use it to reason about correctness and insert the appropriate order-preserving mechanisms. However, this is a very low level interface for a programmer: Parallel programming is difficult enough without having to think about reorderings and write atomicity! The specific reorderings and order-enforcing mechanisms supported are different across systems, compromising portability. Ideally, a programmer would assume SC semantics, and the system components would automatically determine what program orders they may alter without violating this illusion. However, this is too difficult to compute. What a programmer therefore wants is a methodology for writing "safe" programs. This is a contract such that if the program follows certain high-level rules or provides enough program annotations—such as telling the system that `flag` in Figure 9-3 is in fact used as a synchronization variable—then any system on which the program runs will always guarantee a sequentially consistent execution, regardless of the default reorderings in the system specifications it supports. The programmer's responsibility is to follow the rules and provide the annotations (which hopefully does not involve reasoning at the level of potential reorderings); the system's responsibility is to maintain the illusion of sequential consistency. Typically, this means using the annotations as constraints on compiler reorderings, and also translating the annotations to the right order-preserving mechanisms for the processor at the right places. The implication for programming languages is that they should support the necessary annotations.

> **The translation mechanism**. This translates the programmer's annotations to the interface exported by the system specification.

---

1. Actually, it is possible to further weaken the requirements for correct execution. For example, it is not necessary for stores to A and B to complete before store to Flag is done; it is only necessary that they be complete by the time processor P2 observes that the value of Flag has changed to 1. It turns out, such relaxed models are extremely difficult to implement in hardware, so we do not discuss them here. In software, some of these relaxed models make sense, and we will discuss them in Section 9.3.

We organize our discussion of relaxed consistency models around these three pieces. We first examine different low-level specifications exported by systems and particularly microprocessors, organizing them by the types of reorderings they allow. Section 9.2.2 discusses the programmer's interface or contract and how the programmer might provide the necessary annotations, and Section 9.2.3 briefly discusses translation mechanisms. A detailed treatment of implementation complexity and performance benefits will be postponed until we discuss latency tolerance mechanisms in Chapter 11.

9.2.1 System Specifications

Several different reordering specifications have been proposed by microprocessor vendors and by researchers, each with its own mechanisms for enforcing orders. These include total store ordering (TSO) [SFC91, Spa91], partial store ordering (PSO) [SFC91, Spa91], and relaxed memory ordering (RMO) [WeG94] from the SUN Sparc V8 and V9 specifications, processor consistency described in [Goo89, GLL+90] and used in the Intel Pentium processors, weak ordering (WO) [DSB86,DuS90], release consistency (RC) [GLL+90], and the Digital Alpha [Sit92] and IBM/Motorola PowerPC [MSS+94] models. Of course, a particular implementation of a processor may not support all the reorderings that its system specification allows. The system specification defines the semantic interface for that architecture, i.e. what orderings the programmer must assume might happen; the implementation determines what reorderings actually happen and how much performance can actually be gained.

Let us discuss some of the specifications or consistency models, using the relaxations in program order that they allow as our primary axis for grouping models together [Gha95]. The first set of models only allows a read to bypass (complete before) an earlier incomplete write in program order; i.e. allows the write->read order to be reordered (TSO, PC). The next set allows this and also allows writes to bypass previous writes; i.e. write->write reordering (PSO). The final set allows reads or writes to bypass previous reads as well; that is, allows all reorderings among read and write accesses (WO, RC, RMO, Alpha, PowerPC). Note that a read-modify-write operation is treated as being both a read and a write, so it is reordered with respect to another operation only if both a read and a write can be reordered with respect to that operation. Of course, in all cases we assume basic cache coherence—write propagation and write serialization—and that uniprocessor data and control dependences are maintained within each process. The specifications discussed have in most cases been motivated by and defined for the processor architectures themselves, i.e. the hardware interface. They are applicable to compilers as well, but since sophisticated compiler optimizations require the ability to reorder all types of accesses most compilers have not supported as wide a variety of ordering models. In fact, at the programmer's interface, all but the last set of models have their utility limited by the fact that they do not allow many important compiler optimizations. The focus in this discussion is mostly on the models; implementation issues and performance benefits will be discussed in Chapter 11.

Relaxing the *Write-to-Read* Program Order

The main motivation for this class of models is to allow the hardware to hide the latency of write operations that miss in the first-level cache. While the write miss is still in the write buffer and not yet visible to other processors, the processor can issue and complete reads that hit in its cache or even a single read that misses in its cache. The benefits of hiding write latency can be substantial, as we shall see in Chapter 11, and most processors can take advantage of this relaxation.

How well do these models preserve the programmer's intuition even without any special operations? The answer is quite well, for the most part. For example, spinning on a flag for event synchronization works without modification (Figure 9-4(a)). The reason is that TSO and PC models preserve the ordering of writes, so the write of the flag is not visible until all previous writes have completed in the system. For this reason, most early multiprocessors supported one of these two models, including the Sequent Balance, Encore Multimax, VAX-8800, SparcCenter1000/2000, SGI 4D/240, SGI Challenge, and even the Pentium Pro Quad, and it has been relatively easy to port even complex programs such as the operating systems to these machines.

Of course, the semantics of these models is not SC, so there are situations where the differences show through and SC-based intuition is violated. Figure 9-4 shows four code examples, the first three of which we have seen earlier, where we assume that all variables start out having the value 0. In segment (b), SC guarantees that if B is printed as 1 then A too will be printed as 1, since the

```
 P1              P2
 
 A = 1;          while (Flag==0);
 Flag = 1;       print A;
                (a)
```

```
 P1              P2
 
 A = 1;          print B;
 B = 1;          print A;
                (b)
```

```
 P1         P2               P3
 
 A = 1;     while (A==0);    while (B==0);
            B = 1;           print A;
                    (c)
```

```
 P1              P2
 
 A = 1;    (i)   B = 1;    (iii)
 print B;  (ii)  print A;  (iv)
                (d)
```
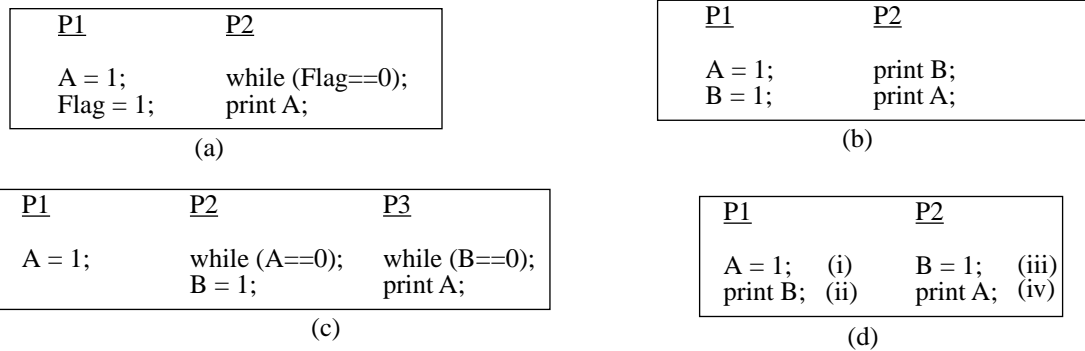
Figure 9-4 Example code sequences repeated to compare TSO, PC and SC.

Both TSO and PC provide same results as SC for code segments (a) and (b), PC can violate SC semantics for segment (c) (TSO still provides SC semantics), and both TSO and PC violate SC semantics for segment (d).

writes of A and B by P1 cannot be reordered. For the same reason, TSO and PC also have the same semantics. For segment (c), only TSO offers SC semantics and prevents A from being printed as 0, not PC. The reason is that PC does not guarantee write atomicity. Finally, for segment (d) under SC no interleaving of the operations can result in 0 being printed for both A and B. To see why, consider that program order implies the precedence relationships (i) -> (ii) and (iii) -> (iv) in the interleaved total order. If B=0 is observed, it implies (ii) -> (iii), which therefore implies (i) -> (iv). But (i) -> (iv) implies A will be printed as 1. Similarly, a result of A=0 implies B=1. A popular software-only mutual exclusion algorithm called Dekker's algorithm—used in the absence of hardware support for atomic read-modify-write operations [TW97]—relies on the property that both A and B will not be read as 0 in this case. SC provides this property, further contributing to its view as an intuitive consistency model. Neither TSO nor PC guarantees it, since they allow the read operation corresponding to the print to complete before previous writes are visible.

To ensure SC semantics when desired (e.g., to port a program written under SC assumptions to a TSO system), we need mechanisms to enforce two types of extra orderings: (i) to ensure that a read does not complete before an earlier write (applies to both TSO and PC), and (ii) to ensure write atomicity for a read operation (applies only to PC). For the former, different processor architectures provide somewhat different solutions. For example, the SUN Sparc V9 specification

[WeG94] provides *memory barrier* (MEMBAR) or *fence* instructions of different flavors that can ensure any desired ordering. Here, we would insert a write-to-read ordering flavored MEMBAR before the read. This MEMBAR flavor prevents a read that follows it in program order from issuing before a write that precedes it has completed. On architectures that do not have memory barriers, it is possible to achieve this effect by substituting an atomic read-modify-write operation or sequence for the original read. A read-modify-write is treated as being both a read and a write, so it cannot be reordered with respect to previous writes in these models. Of course the value written in the read-modify-write must be the same as the value read to preserve correctness. Replacing a read with a read-modify-write also guarantees write atomicity at that read on machines supporting the PC model. The details of why this works are subtle, and the interested reader can find them in the literature [AGG+93].

Relaxing the *Write-to-Read* and *Write-to-Write* Program Orders

Allowing writes as well to bypass previous outstanding writes to different locations allows multiple writes to be merged in the write buffer before updating main memory, and thus multiple write misses to be overlapped and become visible out of order. The motivation is to further reduce the impact of write latency on processor stall time, and to improve communication efficiency between processors by making new data values visible to other processors sooner. SUN Sparc's PSO model [SFC91, Spa91] is the only model in this category. Like TSO, it guarantees write atomicity.

Unfortunately, reordering of writes can violate our intuitive SC semantics quite a bit. Even the use of ordinary variables as flags for event synchronization (Figure 9-4(a)) is no longer guaranteed to work: The write of `flag` may become visible to other processors before the write of `A`. This model must therefore demonstrate a substantial performance benefit to be attractive.

The only additional instruction we need over TSO is one that enforces write-to-write ordering in a process's program order. In SUN Sparc V9, this can be achieved by using a MEMBAR instruction with the write-to-write flavor turned on (the earlier Sparc V8 provided a special instruction called store-barrier or STBAR to achieve this effect). For example, to achieve the intuitive semantics we would insert such an instruction between the writes of `A` and `flag`.

Relaxing *All* Program Orders: Weak Ordering and Release Consistency

In this final class of specifications, no program orders are guaranteed by default (other than data and control dependences within a process, of course). The benefit is that multiple read requests can also be outstanding at the same time, can be bypassed by later writes in program order, and can themselves complete out of order, thus allowing us to hide read latency. These models are particularly useful for dynamically scheduled processors, which can in fact proceed past read misses to other memory references. They are also the only ones that allow many of the reorderings (even elimination) of accesses that are done by compiler optimizations. Given the importance of these compiler optimizations for node performance, as well as their transparency to the programmer, these may in fact be the only reasonable high-performance models for multiprocessors unless compiler analysis of potential violations of consistency makes dramatic advances. Prominent models in this group are weak ordering (WO) [DSB86,DuS90], release consistency (RC) [GLL+90], Digital Alpha [Sit92], Sparc V9 relaxed memory ordering (RMO) [WeG94], and IBM PowerPC [MSS+94,CSB93]. WO is the seminal model, RC is an extension of WO supported by the Stanford DASH prototype [LLJ+93], and the last three are supported in commercial

architectures. Let us discuss them individually, and see how they deal with the problem of providing intuitive semantics despite all the reordering; for instance, how they deal with the flag example.

Weak Ordering: The motivation behind the weak ordering model (also known as the weak consistency model) is quite simple. Most parallel programs use synchronization operations to coordinate accesses to data when this is necessary. Between synchronization operations, they do not rely on the order of accesses being preserved. Two examples are shown in Figure 9-5.The left segment uses a lock/unlock pair to delineate a critical section inside which the head of a linked list is updated. The right segment uses flags to control access to variables participating in a producer-consumer interaction (e.g., A and D are produced by P1 and consumed by P2). The key in the flag example is to think of the accesses to the flag variables as synchronization operations, since that is indeed the purpose they are serving. If we do this, then in both situations the intuitive semantics are not violated by any read or write reorderings that happen between synchronization accesses (i.e. the critical section in segment (a) and the four statements after the while loop in segment (b)). Based on these observations, weak ordering relaxes all program orders by default, and guarantees that orderings will be maintained only at synchronization operations that can be identified by the system as such. Further orderings can be enforced by adding synchronization operations or labeling some memory operations as synchronization. How appropriate operations are identified as synchronization will be discussed soon.

```
P1, P2,..., Pn                            P1                    P2

...                         TOP: while(flag2==0);  TOP: while(flag1==0);
Lock(TaskQ)                      A = 1;                  x = A;
newTask->next = Head;            u = B;                  y = D;
if (Head != NULL)                v = C;                  B = 3;
  Head->prev = newTask;          D = B * C;              C = D / B;
                                 flag2 = 0;              flag1 = 0;
Head = newTask;                  flag1 = 1;              flag2 = 1;
UnLock(TaskQ)                    goto TOP;               goto TOP;
...
        (a)                                      (b)
```
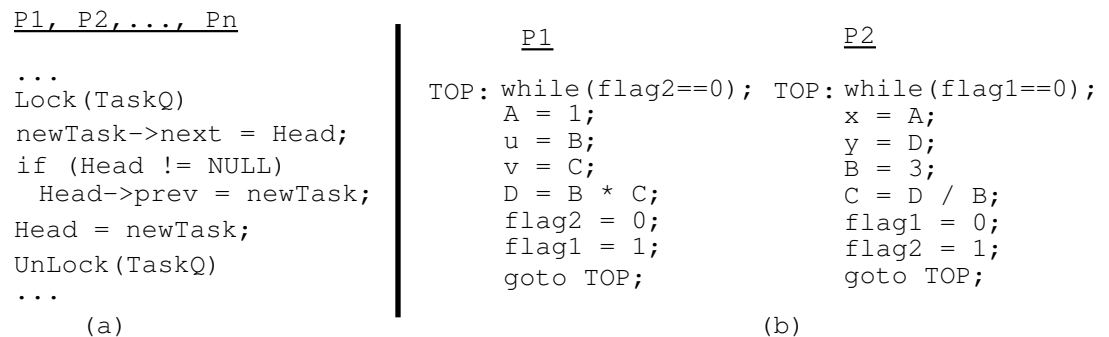
Figure 9-5 Use of synchronization operations to coordinate access to ordinary shared data variables.

The synchronization may be through use of explicit lock and unlock operations (e.g., implemented using atomic test-&-set instructions) or through use of flag variables for producer-consumer sharing.

The left side of Figure 9-6 illustrates the reorderings allowed by weak ordering. Each block with a set of reads/writes represents a contiguous run of non-synchronization memory operations from a processor. Synchronization operations are shown separately. Sufficient conditions to ensure a WO system are as follows. Before a synchronization operation is issued, the processor waits for *all* previous operations in program order (both reads and writes) to have completed. Similarly, memory accesses that follow the synchronization operation are not issued until the synchronization operation completes. Read, write and read-modify-write operations that are not labeled as synchronization can be arbitrarily reordered between synchronization operations. Especially when synchronization operations are infrequent, as in many parallel programs, WO typically provides considerable reordering freedom to the hardware and compiler.
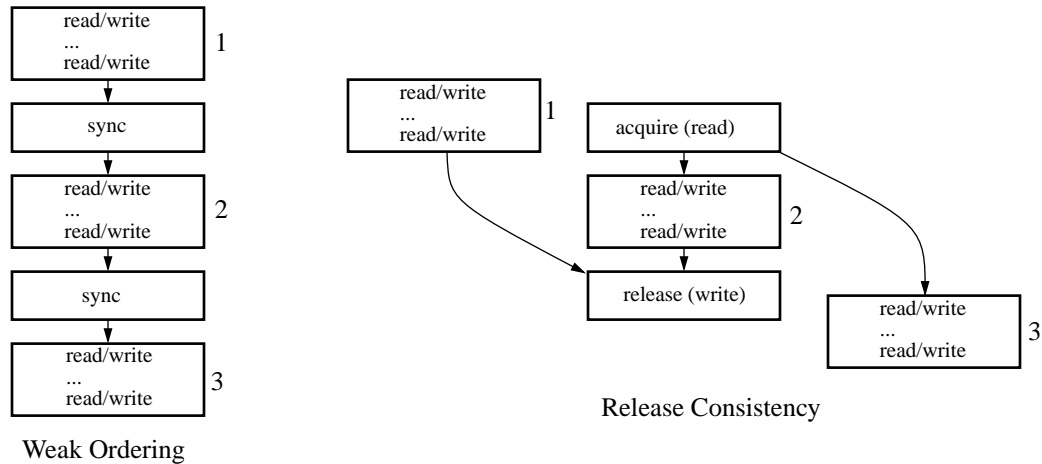
Figure 9-6 Comparison of weak-ordering and release consistency models.

The operations in block 1 precede the first synchronization operation, which is an acquire, in program order. Block 2 is between the two synchronization operations, and block 3 follows the second synchronization operation, which is a release.

Release Consistency: Release consistency observes that weak ordering does not go far enough. It extends the weak ordering model by distinguishing among the types of synchronization operations and exploiting their associated semantics. In particular, it divides synchronization operations into acquires and releases. An *acquire* is a read operation (it can also be a read-modify-write) that is performed to gain access to a set of variables. Examples include the Lock(TaskQ) operation in part (a) of Figure 9-5, and the accesses to flag variables within the while conditions in part (b). A *release* is a write operation (or a read-modify-write) that grants permission to another processor to gain access to some variables. Examples include the "UnLock(TaskQ)" operation in part (a) of Figure 9-5, and the statements setting the flag variables to 1 in part (b).[1]

The separation into acquire and release operations can be used to further relax ordering constraints as shown in Figure 9-6. The purpose of an acquire is to delay memory accesses that follow the acquire statement until the acquire succeeds. It has nothing to do with accesses that precede it in program order (accesses in the block labeled "1"), so there is no reason to wait for those accesses to complete before the acquire can be issued. That is, the acquire itself can be reordered with respect to previous accesses. Similarly, the purpose of a release operation is to grant access to the new values of data modified before it in program order. It has nothing to do with accesses that follow it (accesses in the block labeled "3"), so these need not be delayed until the release has completed. However, we must wait for accesses in block "1" to complete before the release is visible to other processors (since we do not know exactly which variables are associated with the release or that the release "protects"[2]), and similarly we must wait for the acquire

---

1. Exercise: In Figure 9-5 are the statements setting Flag variables to 0 synchronization accesses?

2. It is possible that the release also grants access to variables outside of the variables controlled by the preceding acquire. The exact association of variables with synchronization accesses is very difficult to exploit at the hardware level. Software implementations of relaxed models, however, do exploit such optimizations, as we shall see in Section 9.4.

to complete before the operations in block "3" can be performed. Thus, the sufficient conditions for providing an RC interface are as follows: Before an operation labeled as a release is issued, the processor waits until all previous operations in program order have completed; and operations that follow an acquire operation in program order are not issued until that acquire operation completes. These are sufficient conditions, and we will see how we might be more aggressive when we discuss alternative approaches to a shared address space that rely on relaxed consistency models for performance.

Digital Alpha, Sparc V9 RMO, and IBM PowerPC memory models. The WO and RC models are specified in terms of using labeled synchronization operations to enforce orders, but do not take a position on the exact operations (instructions) that must be used. The memory models of some commercial microprocessors provide no ordering guarantees by default, but provide specific hardware instructions called memory barriers or *fences* that can be used to enforce orderings. To implement WO or RC with these microprocessors, operations that the WO or RC program labels as synchronizations (or acquires or releases) cause the compiler to insert the appropriate special instructions, or the programer can insert these instructions directly.

The Alpha architecture [Sit92], for example, supports two kinds of fence instructions: (i) the

Table 9-1 Characteristics of various system-centric relaxed models. A "yes" in the appropriate column indicates that those orders can be violated by that system-centric model.

Model	W-to-R reorder	W-to-W reorder	R-to-R/W reorder	Read Other's Write Early	Read Own Write Early	Operations for Ordering
SC					yes	
TSO	yes				yes	MEMBAR, RMW
PC	yes			yes	yes	MEMBAR, RMW
PSO	yes	yes			yes	STBAR, RMW
WO	yes	yes	yes		yes	SYNC
RC	yes	yes	yes	yes	yes	REL, ACQ, RMW
RMO	yes	yes	yes		yes	various MEMBARs
Alpha	yes	yes	yes		yes	MB, WMB
PowerPC	yes	yes	yes	yes	yes	SYNC

memory barrier (MB) and (ii) the write memory barrier (WMB). The MB fence is like a synchronization operation in WO: It waits for all previously issued memory accesses to complete before issuing any new accesses. The WMB fence imposes program order only between writes (it is like the STBAR in PSO). Thus, a read issued after a WMB can still bypass a write access issued before the WMB, but a write access issued after the WMB cannot. The Sparc V9 relaxed memory order (RMO) [WeG94] provides a fence instruction with four flavor bits associated with it. Each bit indicates a particular type of ordering to be enforced between previous and following load/store operations (the four possibilities are read-to-read, read-to-write, write-to-read, and write-to-write orderings). Any combinations of these bits can be set, offering a variety of ordering choices. Finally, the IBM PowerPC model [MSS+94,CSB93] provides only a single fence instruction, called SYNC, that is equivalent to Alpha's MB fence. It also differs from the Alpha and RMO models in that the writes are not atomic, as in the processor consistency model. The

model envisioned is WO, to be synthesized by putting SYNC instructions before and after every synchronization operation. You will see how different models can be synthesized with these primitives in Exercise 9.3.

The prominent specifications discussed above are summarized in Table 9-1.[1] They have different performance implications, and require different kinds of annotations to ensure orderings. It is worth noting again that although several of the models allow some reordering of operations, if program order is defined as seen by the programmer then only the models that allow both read and write operations to be reordered within sections of code (WO, RC, Alpha, RMO, and PowerPC) allow the flexibility needed by many important compiler optimizations. This may change if substantial improvements are made in compiler analysis to determine what reorderings are possible given a consistency model. The portability problem of these specifications should also be clear: for example, a program with enough memory barriers to work "correctly" (produce intuitive or sequentially consistent executions) on a TSO system will not necessarily work "correctly" when run on an RMO system: It will need more special operations. Let us therefore examine higher-level interfaces that are convenient for programmers and portable to the different systems, in each case safely exploiting the performance benefits and reorderings that the system affords.

9.2.2 The Programming Interface

The programming interfaces are inspired by the WO and RC models, in that they assume that program orders do not have to be maintained at all between synchronization operations. The idea is for the program to ensure that all synchronization operations are explicitly labeled or identified as such. The compiler or runtime library translates these synchronization operations into the appropriate order-preserving operations (memory barriers or fences) called for by the system specification. Then, the system (compiler plus hardware) guarantees sequentially consistent executions even though it may reorder operations between synchronization operations in any way it desires (without violating dependences to a location within a process). This allows the compiler sufficient flexibility between synchronization points for the reorderings it desires, and also allows the processor to perform as many reorderings as permitted by its memory model: If SC executions are guaranteed even with the weaker models that allow all reorderings, they surely will be guaranteed on systems that allow fewer reorderings. The consistency model presented at the programmer's interface should be at least as weak as that at the hardware interface, but need not be the same.

Programs that label all synchronization events are called *synchronized programs*. Formal models for specifying synchronized programs have been developed; namely the *data-race-free* models influenced by weak ordering [AdH90a] and the *properly-labeled* model [GAG+92] influenced by release consistency [GLL+90]. Interested readers can obtain more details from these references (the differences between them are small). The basic question for the programmer is how to decide which operations to label as synchronization. This is of course already done in the major-

------

1. The relaxation "read own write early" relates to both program order and write atomicity. The processor is allowed to read its own previous write before the write is serialized with respect to other writes to the same location. A common hardware optimization that relies on this relaxation is the processor reading the value of a variable from its own write buffer. This relaxation applies to almost all models without violating their semantics. It even applies to SC, as long as other program order and atomicity requirements are maintained.

ity of cases, when explicit, system-specified primitives such as locks and barriers are used. These are usually also easy to distinguish as acquire or release, for memory models such as RC that can take advantage of this distinction; for example, a lock is an acquire and an unlock is a release, and a barrier contains both since arrival at a barrier is a release (indicates completion of previous accesses) while leaving it is an acquire (obtaining permission for the new set of accesses). The real question is how to determine which memory operations on ordinary variables (such as our flag variables) should be labeled as synchronization. Often, programmers can identify these easily, since they know when they are using this event synchronization idiom. The following definitions describe a more general method when all else fails:

> **Conflicting operations**: Two memory operations from different processes are said to *conflict* if they access the same memory location and at least one of them is a write.

> **Competing operations**: These are a subset of the conflicting operations. Two conflicting memory operations (from different processes) are said to be competing if it is possible for them to appear next to each other in a sequentially consistent total order (execution); that is, to appear one immediately following the other in such an order, with no intervening memory operations on shared data between them.

> **Synchronized Program**: A parallel program is *synchronized* if all competing memory operations have been labeled as synchronization (perhaps differentiated into acquire and release by labeling the read operations as acquires and the write operations as releases).

The fact that competing means competing under any possible SC interleaving is an important aspect of the programming interface. Even though the system uses a relaxed consistency model, the reasoning about where annotations are needed can itself be done while assuming an intuitive, SC execution model, shielding the programmer from reasoning directly in terms of reorderings. Of course, if the compiler could automatically determine what operations are conflicting or competing, then the programmer's task would be made a lot simpler. However, since the known analysis techniques are expensive and/or conservative [ShS88, KrY94], the job is almost always left to the programmer.

Consider the example in Figure 9-7. The accesses to the variable `flag` are competing operations
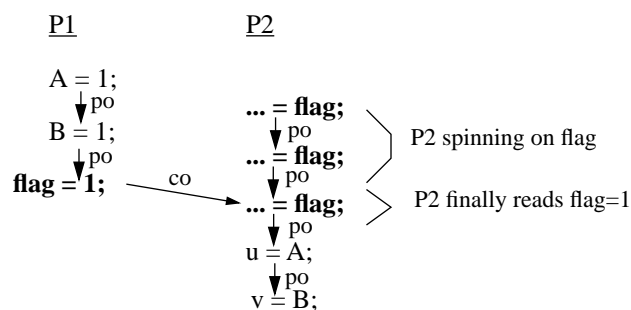
Figure 9-7 An example code sequence using spinning on a flag to synchronize producer-consumer behavior.

The arcs labeled "po" show program order, and that labeled "co" shows conflict order. Notice that between write to A by P1 and read to A by P2, there is a chain of accesses formed by po and co arcs. This will be true in *all* executions of the program. Such chains which have at least one po arc, will not be present for accesses to the variable flag (there is only a co arc).

by the above definition. What this really means is that on a multiprocessor they may execute simultaneously on their respective processors, and we have no guarantee about which executes first. Thus, they are also said to constitute *data races*. In contrast, the accesses to variable A (and

to B) by P1 and P2 are conflicting operations, but they are necessarily separated in any SC interleaving by an intervening write to the variable `flag` by P1 and a corresponding read of `flag` by P2. Thus, they are not competing accesses and do not have to be labeled as synchronization.

To be a little more formal, given a particular SC execution order the *conflict order* (co) is the order in which the conflicting operations occur (note that in a given total or execution order one must occur before the other). In addition, we have the program order (po) for each process. Figure 9-7 also shows the program orders and conflict order for a sample execution of our code fragment. Two accesses are non-competing if under all possible SC executions (interleavings) there always exists a chain of other references between them, such that at least one link in the chain is formed by a program-order rather than conflict order arc. The complete formalism can be found in the literature [Gha95].

Of course, the definition of synchronized programs allows a programmer to conservatively label more operations than necessary as synchronization, without compromising correctness. In the extreme, labeling all memory operations as synchronization always yields a synchronized program. This extreme example will of course deny us the performance benefits that the system might otherwise provide by reordering non-synchronization operations, and will on most systems yield much worse performance than straightforward SC implementations due to the overhead of the order-preserving instructions that will be inserted. The goal is to only label competing operations as synchronization.

In specific circumstances, we may decide not to label some competing accesses as synchronization, since we want to allow data races in the program. Now we are no longer guaranteed SC semantics, but we may know through application knowledge that the competing operations are not being used as synchronization and we do not need such strong ordering guarantees in certain sections of code. An example is the use of the asynchronous rather than red-black equation solver in Chapter 2. There is no synchronization between barriers (sweeps) so within a sweep the read and write accesses to the border elements of a partition are competing accesses. The program will not satisfy SC semantics on a system that allows access reorderings, but this is okay since the solver repeats the sweeps until convergence: Even if the processes sometimes read old values in an iteration and sometimes new (unpredictably), they will ultimately read updated values and make progress toward convergence. If we had labeled the competing accesses, we would have compromised reordering and performance.

The last issue related to the programming interface is how the above labels are to be specified by the programmer. In many cases, this is quite stylized and already present in the programming language. For example, some parallel programming languages (e.g., High-Performance Fortran [HPF93] allow parallelism to be expressed in only stylized ways from which it is trivial to extract the relevant information. For example, in FORALL loops (loops in which all iterations are independent) only the implicit barrier at the end of the loop needs to be labeled as synchronization: The FORALL specifies that there are no data races within the loop body. In more general programming models, if programmers use a library of synchronization primitives such as LOCK, UNLOCK and BARRIER, then the code that implements these primitives can be labeled by the designer of the library; the programmer needn't do anything special. Finally, if the application programmer wants to add further labels at memory operations—for example at flag variable accesses or to preserve some other orders as in the examples of Figure 9-4 on page 621—we need programming language or library support. A programming language could provide an attribute for variable declarations that indicates that all references to this variable are synchronization accesses, or there could be annotations at the statement level, indicating that a particular

access is to be labeled as a synchronization operation. This tells the compiler to constrain its reordering across those points, and the compiler in turn translates these references to the appropriate order-preserving mechanisms for the processor.

9.2.3 Translation Mechanisms

For most microprocessors, translating labels to order preserving mechanisms amounts to inserting a suitable memory barrier instruction before and/or after each operation labeled as a synchronization (or acquire or release). It would save instructions if one could have flavor bits associated with individual loads/stores themselves, indicating what orderings to enforce and avoiding extra instructions, but since the operations are usually infrequent this is not the direction that most microprocessors have taken so far.

9.2.4 Consistency Models in Real Systems

With the large growth in sales of multiprocessors, modern microprocessors are designed so that they can be seamlessly integrated into these machines. As a result, microprocessor vendors expend substantial effort defining and precisely specifying the memory model presented at the hardware-software interface. While sequential consistency remains the best model to reason about correctness and semantics of programs, for performance reasons many vendors allow orders to be relaxed. Some vendors like Silicon Graphics (in the MIPS R10000) continue to support SC by allowing out-of-order issue and execution of operations, but not out of order completion or visibility. This allows overlapping of memory operations by the dynamically scheduled processor, but forces them to complete in order. The Intel Pentium family supports a processor consistency model, so reads can complete before previous writes in program order (see illustration), and many microprocessors from SUN Microsystems support TSO which allows the same reorderings. Many other vendors have moved to models that allow all orders to be relaxed (e.g., Digital Alpha and IBM PowerPC) and provide memory barriers to enforce orderings where necessary.

At the hardware interface, multiprocessors usually follow the consistency model exported by the microprocessors they use, since this is the easiest thing to do. For example, we saw that the NUMA-Q system exports the processor consistency model of its Pentium Pro processors at this interface. In particular, on a write the ownership and/or data are obtained before the invalidations begin. The processor is allowed to complete its write and go on as soon as the ownership/data is received, and the SCLIC takes care of the invalidation/acknowledgment sequence. It is also possible for the communication assist to alter the model, within limits, but this comes at the cost of design complexity. We have seen an example in the Origin2000, where preserving SC requires that the assist (Hub) only reply to the processor on a write once the exclusive reply and all invalidation acknowledgments have been received (called *delayed* exclusive replies). The dynamically scheduled processor can then retire its write from the instruction lookahead buffer and allow subsequent operations to retire and complete as well. If the Hub replies as soon as the exclusive reply is received and before invalidation acknowledgments are received (called *eager* exclusive replies), then the write will retire and subsequent operations complete before the write is actually completed, and the consistency model is more relaxed. Essentially, the Hub fools the processor about the completion of the write.

Having the assist cheat the processor can enhance performance but increases complexity, as in the case of eager exclusive replies. The handling of invalidation acknowledgments must now be

done asynchronously by the Hub through tracking buffers in its processor interface, in accordance with the desired relaxed consistency model. There are also questions about whether subsequent accesses to a block from other processors, forwarded to this processor from the home, should be serviced while invalidations for a write on that block are still outstanding (see Exercise 9.2), and what happens if the processor has to write that block back to memory due to replacement while invalidations are still outstanding in the Hub. In the latter case, either the writeback has to be buffered by the Hub and delayed till all invalidations are acknowledged, or the protocol must be extended so a later access to the written back block is not satisfied by the home until the acknowledgments are received by the requestor. The extra complexity and the perceived lack of substantial performance improvement led the Origin designers to persist with a sequential consistency model, and reply to the processor only when all acknowledgments are received.

On the compiler side, the picture for memory consistency models is not so well defined, complicating matters for programmers. It does not do a programmer much good for the processor to support sequential consistency or processor consistency if the compiler reorders accesses as it pleases before they even get to the processor (as uniprocessor compilers do). Microprocessor memory models are defined at the hardware interface; they tend to be concerned with program order as presented to the processor, and assume that a separate arrangement will be made with the compiler. As we have discussed, exporting intermediate models such as TSO, processor consistency and PSO up to the programmer's interface does not allow the compiler enough flexibility for reordering. We might assume that the compiler most often will not reorder the operations that would violate the consistency model, e.g. since most compiler reorderings of memory operations tend to focus on loops, but this is a very dangerous assumption, and sometimes the orders we rely upon indeed occur in loops. To really use these models at the programmer's interface, uniprocessor compilers would have to be modified to follow these reordering specifications, compromising performance significantly. An alternative approach, mentioned earlier, is to use compiler analysis to determine when operations can be reordered without compromising the consistency model exposed to the user [ShS88, KrY94]. However, the compiler analysis must be able to detect conflicting operations from different processes, which is expensive and currently rather conservative.

More relaxed models like Alpha, RMO, PowerPC, WO, RC and synchronized programs can be used at the programmer's interface because they allow the compiler the flexibility it needs. In these cases, the mechanisms used to communicate ordering constraints must be heeded not only by the processor but also by the compiler. Compilers for multiprocessors are beginning to do so (see also Section 9.6). Underneath a relaxed model at the programmer's interface, the processor interface can use the same or a stronger ordering model, as we saw in the context of synchronized programs. However, there is now significant motivation to use relaxed models even at the processor interface, to realize the performance potential. As we move now to discussing alternative approaches to supporting a shared address space with coherent replication of data, we shall see that relaxed consistency models can be critical to performance when we want to support coherence at larger granularities than cache blocks. We will also see that the consistency model can be relaxed even beyond release consistency (can you think how?).

9.3 Overcoming Capacity Limitations

The systems that we have discussed so far provide automatic replication and coherence in hardware only in the processor caches. A processor cache replicates remotely allocated data directly

upon reference, without it being replicated in the local main memory first. Access to main memory has nonuniform cost depending on whether the memory is local or remote, so these systems are traditionally called *cache coherent non-uniform memory access (CC-NUMA)* architectures. On a cache miss, the assist determines from the physical address whether to look up local memory and directory state or to send the request directly to a remote home node. The granularity of communication, of coherence, and of allocation in the replication store (cache) is a cache block. As discussed earlier, a problem with these systems is that the capacity for local replication is limited to the hardware cache. If a block is replaced from the cache, it must be fetched from remote memory if it is needed again, incurring artifactual communication. The goal of the systems discussed in this section is to overcome the replication capacity problem while still providing coherence in hardware and at fine granularity of cache blocks for efficiency.

9.3.1 Tertiary Caches

There are several ways to achieve this goal. One is to use a large but slower remote access cache, as in the Sequent NUMA-Q and Convex Exemplar [Con93,TSS+96]. This may be needed for functionality anyway if the nodes of the machine are themselves small-scale multiprocessors—to keep track of remotely allocated blocks that are currently in the local processor caches—and can simply be made larger for performance. It will then also hold replicated remote blocks that have been replaced from local processor caches. In NUMA-Q, this DRAM remote cache is at least 32MB large while the sum of the four lowest-level processor caches in a node is only 2 MB. A similar method, which is sometimes called the tertiary cache approach, is to take a fixed portion of the local main memory and manage it like a remote cache.

These approaches replicate data in main memory at fine grain, but they do not automatically *migrate* or change the home of a block to the node that incurs cache misses most often on that block. Space is always allocated for the block in the regular part of main memory at the original home. Thus, if data were not distributed appropriately in main memory by the application, then in the tertiary cache approach even if only one processor ever accesses a given memory block (no need for multiple copies or replication) the system may end up wasting half the available main memory in the system: There are two copies of each block, but only one is ever used. Also, a statically established tertiary cache is wasteful if its replication capacity is not is not needed for performance. The other approach to increasing replication capacity, which is called *cache-only memory architecture* or *COMA*, is to treat all of local memory as a hardware-controlled cache. This approach, which achieves both replication and migration, is discussed in more depth next. In all these cases, replication and coherence are managed at a fine granularity, though this does not necessarily have to be the same as the block size in the processor caches.

9.3.2 Cache-only Memory Architectures (COMA)

In COMA machines, every fine-grained memory block in the entire main memory has a hardware tag associated with it. There is no fixed node where there is always guaranteed to be space allocated for a memory block. Rather, data dynamically migrate to or are replicated at the main memories of the nodes that access or "attract" them; these main memories organized as caches are therefore called *attraction memories*. When a remote block is accessed, it is replicated in attraction memory as well as brought into the cache, and is kept coherent in both places by hardware. Since a data block may reside in any attraction memory and may move transparently from one to the other, the location of data is decoupled from its physical address and another addressing mechanism must be used. Migration of a block is obtained through replacement in the attrac-

tion memory: If block *x* originally resides in node *A*'s main (attraction) memory, then when node *B* reads it B will obtain a copy (replication); if the copy in *A*'s memory is later replaced by another block that *A* references, then the only copy of that block left is now in *B*'s attraction memory. Thus, we do not have the problem of wasted original copies that we potentially had with the tertiary cache approach. Automatic data migration also has substantial advantages for multi-programmed workloads in which the operating system may decide to migrate processes among nodes at any time.

Hardware-Software Tradeoffs

The COMA approach introduces clear hardware-software tradeoffs (as do the other approaches). By overcoming the cache capacity limitations of the pure CC-NUMA approach, the goal is to free parallel software from worrying about data distribution in main memory. The programmer can view the machine as if it had a centralized main memory, and worry only about inherent communication and false sharing (of course, cold misses may still be satisfied remotely). While this makes the task of software writers much easier, COMA machines require a lot more hardware support than pure CC-NUMA machines to implement main memory as a hardware cache. This includes per-block tags and state in main memory, as well as the necessary comparators. There is also the extra memory overhead needed for replication in the attraction memories, which we shall discuss shortly. Finally, the coherence protocol for attraction memories is more complicated than that we saw for processor caches. There are two reasons for this, both having to do with the fact that data move dynamically to where they are referenced and do not have a fixed "home" to back them up. First, the location of the data must be determined upon an attraction memory miss, since it is no longer bound to the physical address. Second, with no space necessarily reserved for the block at the home, it is important to ensure that the last or only copy of a block is not lost from the system by being replaced from its attraction memory. This extra complexity is not a problem in the tertiary cache approach.

Performance Tradeoffs

Performance has its own interesting set of tradeoffs. While the number of remote accesses due to artifactual communication is reduced, COMA machines tend to increase the latency of accesses that do need to be satisfied remotely, including cold, true sharing and false sharing misses. The reason is that even a cache miss to nonlocal memory needs to first look up the local attraction memory, to see if it has a local copy of the block. The attraction memory access is also a little more expensive than a standard DRAM access because the attraction memory is usually implemented to be set-associative, so there may be a tag selection in the critical path.

In terms of performance, then, COMA is most likely to be beneficial for applications that have high capacity miss rates in the processor cache (large working sets) to data that are not allocated locally to begin with, and most harmful to applications where performance is dominated by communication misses. The advantages are also greatest when access patterns are unpredictable or spatially interleaved from different processes at fine grain, so data placement or replication/migration at page granularity in software would be difficult. For example, COMA machines are likely to be more advantageous when a two-dimensional array representation is used for a near-neighbor grid computation than when a four-dimensional array representation is used, because in the latter case appropriate data placement at page granularity is not difficult in software. The higher cost of communication may in fact make them perform worse than pure CC-NUMA when four-dimensional arrays are used with data placement. Figure 9-8 summarizes the tradeoffs in

terms of application characteristics. Let us briefly look at some design options for COMA proto-

Application Characteristics

Very low miss rate

Should perform similarly
- Dense Linear Algebra
- N-body

Mostly coherence misses

COMA may be worse
due to added latency
- small problem sizes
for many applications
(e.g. FFT, Ocean)
- applications with a lot
of false sharing

Mostly capacity misses

COMA has potential advantage due
to automatic replication/migration

Coarse-grained
data access

Replication and particularly
migration/initial-placement
may be easy at page granularity
even without COMA support
- many regular scientific applications
(Ocean, FFT)

Fine-grained
data access

COMA advantageous
- applications with large
working sets but irregular data
structures and access patterns
(Raytrace with large
data sets)

Figure 9-8 Performance tradeoffs between COMA and non-COMA architectures.

cols, and how they might solve the architectural problems of finding the data on a miss and not losing the last copy of a block.

Design Options: Flat versus Hierarchical Approaches

COMA machines can be built with hierarchical or with flat directory schemes, or even with hierarchical snooping (see Section 8.11.2). Hierarchical directory-based COMA was used in the Data Diffusion Machine prototype [HLH92,Hag92], and hierarchical snooping COMA was used in commercial systems from Kendall Square Research [FBR93]. In these *hierarchical COMA* schemes, data are found on a miss by traversing the hierarchy, just as in regular hierarchical directories in non-COMA machines. The difference is that while in non-COMA machines there is a fixed home node for a memory block in a processing node, here there is not. When a reference misses in the local attraction memory, it proceeds up the hierarchy until a node is found that indicates the presence of the block in its subtree in the appropriate state. The request then proceeds down the hierarchy to the appropriate processing node (which is at a leaf), guided by directory lookups or snooping at each node along the way.

In *flat COMA* schemes, there is still no home for a memory block in the sense of a reserved location for the data, but there is a fixed home where just the directory information can be found

[SJG92,Joe95]. This fixed home is determined from either the physical address or a global identifier obtained from the physical address, so the directory information is also decoupled from the actual data. A miss in the local attraction memory goes to the directory to look up the state and copies, and the directory may be maintained in either a memory-based or cache-based way.The tradeoffs for hierarchical directories versus flat directories are very similar to those without COMA (see Section 8.11.2).

Let us see how hierarchical and flat schemes can solve the last copy replacement problem. If the block being replaced is in shared state, then if we are certain that there is another copy of the block in the system we can safely discard the replaced block. But for a block that is in exclusive state or is the last copy in the system, we must ensure that it finds a place in some other attraction memory and is not thrown away. In the hierarchical case, for a block in shared state we simply have to go up the hierarchy until we find a node that indicates that a copy of the block exists somewhere in its subtree. Then, we can discard the replaced block as long as we have updated the state information on the path along the way. For a block in exclusive state, we go up the hierarchy until we find a node that has a block in invalid or shared state somewhere in its subtree, which this block can replace. If that replaceable block found is in shared rather than invalid state, then its replacement will require the previous procedure.

In a flat COMA, more machinery is required for the last copy problem. One mechanism is to label one copy of a memory block as the *master* copy, and to ensure that the master copy is not dropped upon replacement. A new cache state called *master* is added in the attraction memory. When data are initially allocated, every block is a master copy. Later, a master copy is either an exclusive copy or one of the shared copies. When a shared copy of a block that is not the master copy is replaced, it can be safely dropped (we may if we like send a replacement hint to the directory entry at the home). If a master copy is replaced, a replacement message is sent to the home. The home then chooses another node to send this master to in the hope of finding room, and sets that node to be the master. If all available blocks in that set of the attraction memory cache at this destination are masters, then the request is sent back to the home which tries another node, and so on. Otherwise, one of the replaceable blocks in the set is replaced and discarded (some optimizations are discussed in [JoH94]).

Regardless of whether a hierarchical or flat COMA organization is used, the initial data set of the application should not fill the entire main or attraction memory, to ensure that there is space in the system for a last copy to find a new residence. To find replaceable blocks, the attraction memories should be quite highly associative as well. Not having enough extra (initially unallocated) memory can cause performance problems for several reasons. First, the less the room for replication in the attraction memories, the less effective the COMA nature of the machine in satisfying cache capacity misses locally. Second, little room for replication implies that the replicated blocks are more likely to be replaced to make room for replaced last copies. And third, the traffic generated by replaced last copies can become substantial when the amount of extra memory is small, which can cause a lot of contention in the system. How much memory should be set aside for replication and how much associativity is needed should be determined empirically [JoH94].

Summary: Path of a Read Operation

Consider a flat COMA scheme. A virtual address is first translated to a physical address by the memory management unit. This may cause a page fault and a new mapping to be established, as in a uniprocessor, though the actual data for the page is not loaded into memory. The physical

address is used to look up the cache hierarchy. If it hits, the reference is satisfied. If not, then it must look up the local attraction memory. Some bits from the physical address are used to find the relevant set in the attraction memory, and the tag store maintained by the hardware is used to check for a tag match. If the block is found in an appropriate state, the reference is satisfied. If not, then a remote request must be generated, and the request is sent to the home determined from the physical address. The directory at the home determines where to forward the request, and the owner node uses the address as an index into its own attraction memory to find and return the data. The directory protocol ensures that states are maintained correctly as usual.

9.4 Reducing Hardware Cost

The last of the major issues discussed in this chapter is hardware cost. Reducing cost often implies moving some functionality from specialized hardware to software running on existing or commodity hardware. In this case, the functionality in question is managing for replication and coherence. Since it is much easier for software to control replication and coherence in main memory rather than in the hardware cache, the low-cost approaches tend to provide replication and coherence in main memory as well. The differences from COMA, then, are the higher over-head of communication, and potentially the granularity at which replication and coherence are managed.

Consider the hardware cost of a pure CC-NUMA approach. The portion of the communication architecture that is on a node can be divided into four parts: the part of the assist that checks for access control violations, the per-block tags that it uses for this purpose, the part that does the actual protocol processing (including intervening in the processor cache), and the network inter-face itself. To keep data coherent at cache block granularity, the access control part needs to see every load or store to shared data that misses in the cache so it can take the necessary protocol action. Thus, for fine-grained access control to be done in hardware, the assist must at least be able to snoop on the local memory system (as well as issue requests to it in response to incoming requests from the network).

For coherence to be managed efficiently, each of the other functional components of the assist can benefit greatly from hardware specialization and integration. Determining access faults (i.e. whether or not to invoke the protocol) quickly requires that the tags per block of main memory be located close to this part of the assist. The speed with which protocol actions can be invoked and the assist can intervene in the processor cache increases as the assist is integrated closer to the cache. Performing protocol operations quickly demands that the assist be either hard-wired, or if programmable (as in then Sequent NUMA-Q) then specialized for the types of operations that protocols perform most often (for example bit-field extractions and manipulations). Finally, mov-ing small pieces of data quickly between the assist and network interface asks that the network interface be tightly integrated with the assist. Thus, for highest performance we would like that these four parts of the communication architecture be tightly integrated together, with as few bus crossings as possible needed to communicate among them, and that this whole communication architecture be specialized and tightly integrated into the memory system.

Early cache-coherent machines such as the KSR-1 [FBR93] accomplished this by integrating a hard-wired assist into the cache controller and integrating the network interface tightly into the assist. However, modern processors tend to have even their second-level cache controllers on the processor chip, so it is increasingly difficult to integrate the assist into this controller once the

processor is built. The SGI Origin therefore integrates its hard-wired Hub into the memory controller (and the network interface tightly with the Hub), and the Stanford Flash integrates its specialized programmable protocol engine into the memory controller. These approaches use specialized, tightly integrated hardware support for cache coherence, and do not leverage inexpensive commodity parts for the communication architecture. They are therefore expensive, more so in design and implementation time than in the amount of actual hardware needed.

Research efforts are attempting to lower this cost, with several different approaches. One approach is to perform access control in specialized hardware, but delegate much of the other activity to software and commodity hardware. Other approaches perform access control in software as well, and are designed to provide a coherent shared address space abstraction on commodity nodes and networks with no specialized hardware support. The software approaches provide access control either by instrumenting the program code, or by leveraging the existing virtual memory support to perform access control at page granularity, or by exporting an object-based programming interface and providing access control in a runtime layer at the granularity of user-defined objects.

Let us discuss each of these approaches, which are currently all at the research stage. More time will be spent on the so-called page-based shared virtual memory approach, because it changes the granularity at which data are allocated, communicated and kept coherent while still preserving the same programming interface as hardware-coherent systems, and because it relies on relaxed memory consistency for performance and requires substantially different protocols.

9.4.1 Hardware Access Control with a Decoupled Assist

While specialized hardware support is used for fine-grained access control in this approach, some or all of the other aspects (protocol processing, tags, and network interface) are decoupled from one another. They can then either use commodity parts attached to less intrusive parts of the node like the I/O bus, or use no extra hardware beyond that on the uniprocessor system. For example, the per-block tags and state can be kept in specialized fast memory or in regular DRAM, and protocol processing can be done in software either on a separate, inexpensive general-purpose processor or even on the main processor itself. The network interface usually has some specialized support for fine-grained communication in all these cases, since otherwise the end-point overheads would dominate. Some possible combinations for how the various functions might be integrated are shown in Figure 9-9.

The problem with the decoupled hardware approach, of course, is that it increases the latency of communication, since the interaction of the different components with each other and with the node is slower (e.g. it may involve several bus crossings). More critically, the effective occupancy of the decoupled communication assist is much larger than that of a specialized, integrated assist, which can hurt performance substantially for many applications as we saw in the previous chapter (Section 8.8).

9.4.2 Access Control through Code Instrumentation

It is also possible to use no additional hardware support at all, but perform all the functions needed for fine-grained replication and coherence in main memory in software. The trickiest part of this is fine-grained access control in main memory, for which a standard uniprocessor does not provide support. To accomplish this in software, individual read and write operations can be

(a) AC, NI, and PP integrated together
   and into memory controller (SGI Origin)

(b) AC, NI, and PP integrated together
   on memory bus (Wisc. Typhoon design)

(c) AC and NI integrated together and
on memory bus; separate commodity PP
also on bus (Wisc. Typhoon-1 design)

(d) Separate AC, commodity NI and
   commodity PP on memory bus
   (Wisc. Typhoon-0)
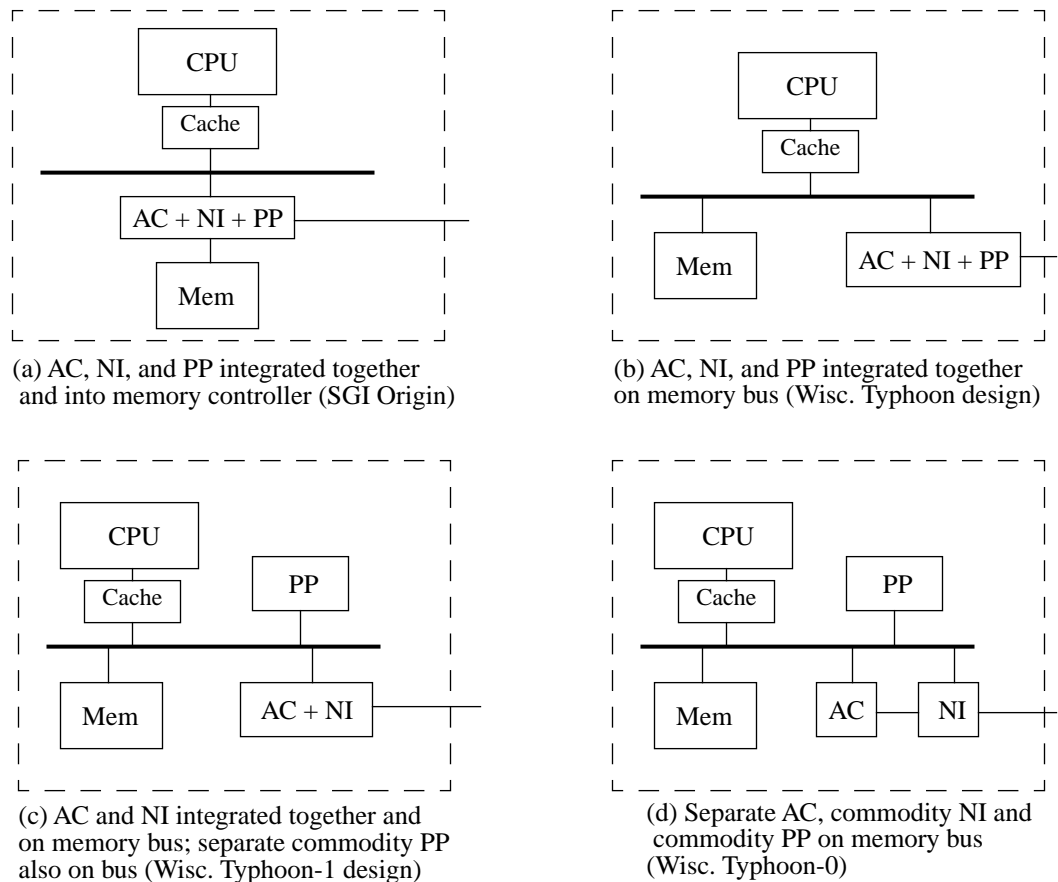
Figure 9-9 Some alternatives for reducing cost over the highly integrated solution.

AC is the access control facility, NI is the network interface, and PP is the protocol processing facility (whether hardwired finite state machine or programmable). The highly integrated solution is shown in (a). The more the parts are separated out, the more the expensive bus crossings needed for them to communicate with one another to process a transaction. The alternatives we have chosen are the Typhoon-0 research prototype from the University of Wisconsin and two other proposals. In these designs, the commodity PP in (b) and (c) is a complete processor like the main CPU, with its own cache system.

instrumented in software to look up per-block tag and state data structures maintained in main memory [SFL+94,SGT96]. To the extent that cache misses can be predicted, only the reads and writes that miss in the processor cache hierarchy need to be thus instrumented. The necessary protocol processing can be performed on the main processor or on whatever form of communication assist is provided. In fact, such software instrumentation allows us to provide access control and coherence at any granularity chosen by the programmer and system, even different granularities for different data structures.

Software instrumentation incurs a run-time cost, since it inserts extra instructions into the code to perform the necessary checks. The approaches that have been developed use several tricks to reduce the number of checks and lookups needed [SGT96], so the cost of access control in this approach may well be competitive with that in the decoupled hardware approach. Protocol pro-

cessing in software on the main processor also has a significant cost, and the network interface and interconnect in these systems are usually commodity-based and hence less efficient than in tightly-coupled multiprocessors.

9.4.3 Page-based Access Control: Shared Virtual Memory

Another approach to providing access control and coherence with no additional hardware support is to leverage the virtual memory support provided by the memory management units of microprocessors and the operating system. Memory management units already perform access control in main memory at the granularity of pages, for example to detect page faults, and manage main memory as a fully associative cache on the virtual address space. By embedding a coherence protocol in the page fault handlers, we can provide replication and coherence at page granularity [LiH89], and manage the main memories of the nodes as coherent, fully associative caches on a shared virtual address space. Access control now requires no special tags, and the assist needn't even see every cache miss. Data enter the local cache only when the corresponding page is already present in local memory. Like in the previous two approaches the processor caches do not have to be kept coherent by hardware, since when a page is invalidated the TLB will not let the processor access its blocks in the cache.

This approach is called *page-based shared virtual memory* or *SVM* for short. Since the cost can be amortized over a whole page of data, protocol processing is often done on the main processor itself, and we can more easily do without special hardware support for communication in the network interface. Thus, there is less need for hardware assistance beyond that available on a standard uniprocessor system.

A very simple form of shared virtual memory coherence is illustrated in Figure 9-10, following an invalidation protocol very similar to those in pure CC-NUMA. A few aspects are worthy of note. First, since the memory management units of different processors manage their main memories independently, the physical address of the page in P1's local memory may be completely different from that of the copy in P0's local memory, even though the pages have the same (shared) virtual address. Second, a page fault handler that implements the protocol must know or determine from where to obtain the up-to-date copy of a page that it has missed on, or what pages to invalidate, before it can set the page's access rights as appropriate and return control to the application process. A directory mechanism can be used for this—every page may have a home, determined by its virtual address, and a directory entry maintained at the home—though high-performance SVM protocols tend to be more complex as we shall see.

The problems with page-based shared virtual memory are (i) the high overheads of protocol invocation and processing and (ii) the large granularity of coherence and communication. The former is expensive because most of the work is done in software on a general-purpose uniprocessor. Page faults take time to cause an interrupt and to switch into the operating system and invoke a handler; the protocol processing itself is done in software, and the messages sent to other processors use the underlying message passing mechanisms that are expensive. On a representative SVM system in 1997, the round-trip cost of satisfying a remote page fault ranges from a few hundred microseconds with aggressive system software support to over a millisecond. This should be compared with less than a microsecond needed for a read miss on aggressive hardware-coherent systems. In addition, since protocol processing is typically done on the main processor (due to the goal of using commodity uniprocessor nodes with no additional hardware support), even
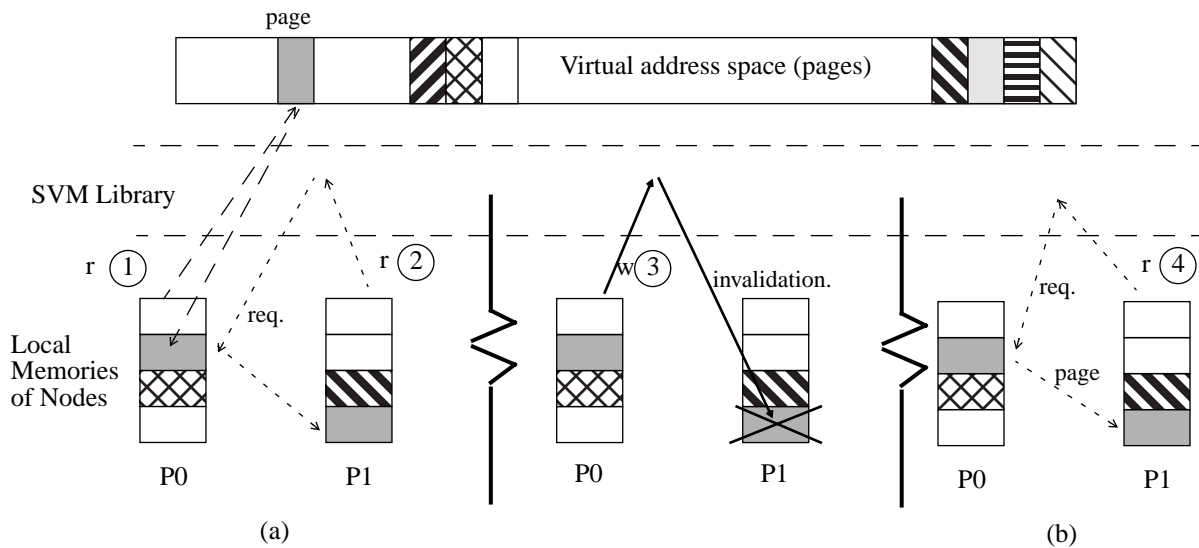
Figure 9-10 Illustration of Simple Shared Virtual Memory.

At the beginning, no node has a copy of the stippled shared virtual page with which we are concerned. Events occur in the order 1, 2, 3, 4. Read 1 incurs a page fault during address translation and fetches a copy of the page to P0 (presumably from disk). Read 2, shown in the same frame, incurs a page fault and fetches a read-only copy to P1 (from P0). This is the same virtual page, but is at two different physical addresses in the two memories. Write 3 incurs a page fault (write to a read-only page), and the SVM library in the page fault handlers determines that P1 has a copy and causes it to be invalidated. P0 now obtains read-write access to the page, which is like the modified or dirty state. When Read 4 by P1 tries to read a location on the invalid page, it incurs a page fault and fetches a new copy from P0 through the SVM library.

incoming requests interrupt the processor, pollute the cache and slow down the currently running application thread that may have nothing to do with that request.

The large granularity of communication and coherence is problematic for two reasons. First, if spatial locality is not very good it causes a lot of fragmentation in communication (only a word is needed but a whole page is fetched). Second, it can easily lead to false sharing, which causes these expensive protocol operations to be invoked frequently. Under a sequential consistency model, invalidations are propagated and performed as soon as the corresponding write is detected, so pages may be frequently ping-ponged back and forth among processors due to either true or false sharing (Figure 9-11 shows an example). This is extremely undesirable in SVM systems due to the high cost of the operations, and it is very important that the effects of false sharing be alleviated.

Using Relaxed Memory Consistency

The solution is to exploit a relaxed memory consistency model such as release consistency, since this allows coherence actions such as invalidations or updates, collectively called *write notices*, to be postponed until the next synchronization point (writes do not have to become visible until then). Let us continue to assume an invalidation-based protocol. Figure 9-12 shows the same example as Figure 9-11 above: The writes to x by processor P1 will not generate invalidations to the copy of the page at P0 until the barrier is reached, so the effects of false sharing will be greatly mitigated and none of the reads of y by P0 before the barrier will incur page faults. Of
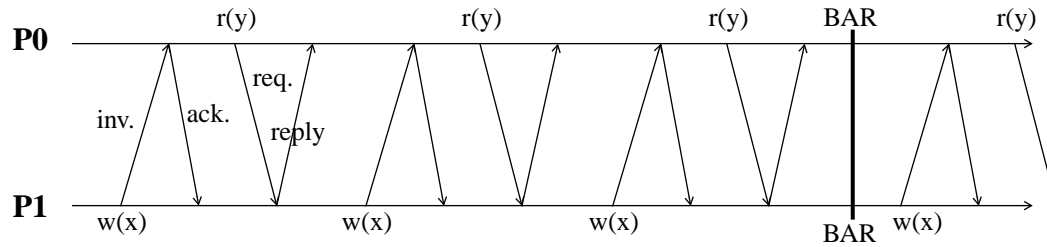
Figure 9-11 Problem with sequential consistency for SVM.

Process P0 repeatedly reads variable y, while process P1 repeatedly writes variable x which happens to fall on the same page as y. Since P1 cannot proceed until invalidations are propagated and acknowledged under SC, the invalidations are propagated immediately and substantial (and very expensive) communication ensues due to this false sharing.
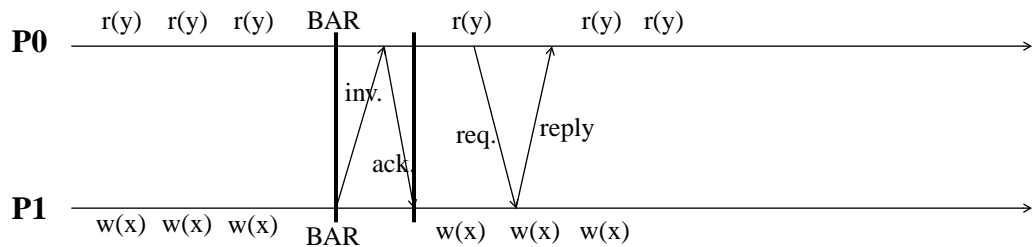
Figure 9-12 Reducing SVM communication due to false sharing with a relaxed consistency model.

There is not communication at reads and writes until a synchronization event, at which point invalidations are propagated to make pages coherent. Only the first access to an invalidated page after a synchronization point generates a page fault and hence request.

course, if P0 accesses y after the barrier it will incur a page fault due to false sharing since the page has now been invalidated.

There is a significant difference from how relaxed consistency is typically used in hardware-coherent machines. There, it is used to avoid stalling the processor to wait for acknowledgments (completion), but the invalidations are usually propagated and applied as soon possible, since this is the natural thing for hardware to do. Although release consistency does not guarantee that the effects of the writes will be seen until the synchronization, in fact they usually will be. The effects of false sharing cache blocks are therefore not reduced much even within a period with no synchronization, and nor is the number of network transactions or messages transferred; the goal is mostly to hide latency from the processor. In the SVM case, the system takes the contract literally: Invalidations are actually not propagated until the synchronization points. Of course, this makes it critical for correctness that all synchronization points be clearly labeled and communicated to the system. [1]

When should invalidations (or write notices) be propagated from the writer to other copies, and when should they be applied. One possibility is to propagate them when the writer issues a release operation. At a release, invalidations for each page that the processor wrote since its pre-

(a) Eager Release Consistency

(b) Lazy Release Consistency
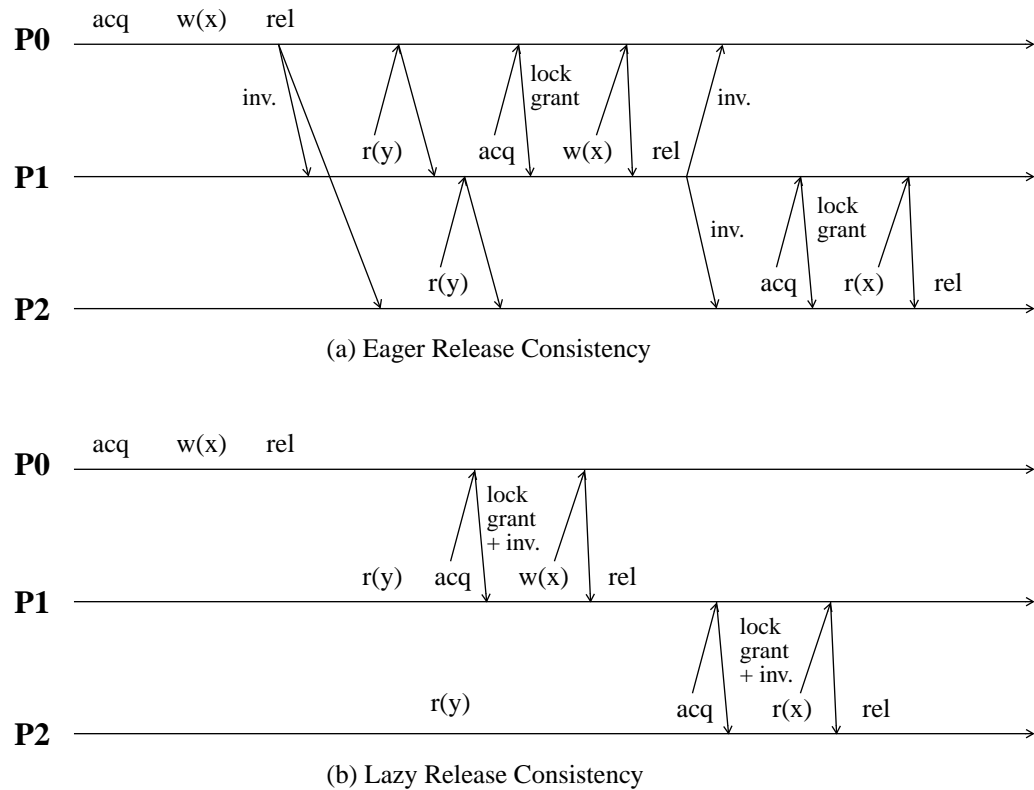
Figure 9-13 Eager versus lazy implementations of release consistency.

The former performs consistency actions (invalidation) at a release point, while the latter performs them at an acquire. Variables x and y are on the same page. The reduction in communication can be substantial, particular for SVM.

vious release are propagated to all processors that have copies of the page. If we wait for the invalidations to complete, this satisfies the sufficient conditions for RC that were presented earlier. However, even this is earlier than necessary: Under release consistency, a given process does not really need to see the write notice until it itself does an acquire. Propagating and applying write notices to all copies at release points, called *eager release consistency* (ERC) [CBZ91,CBZ95], is conservative because it does not know when the next acquire by another processor will occur. As shown in Figure 9-13(a), it can send expensive invalidation messages to more processors than necessary (P2 need not have been invalidated by P0), it requires separate messages for invalidations and lock acquisitions, and it may invalidate processors earlier than necessary thus causing false sharing (see the false sharing between variables x and y, as a result

_____

1. In fact, a similar approach can be used to reduce the effects of false sharing in hardware as well. Invalidations may be buffered in hardware at the requestor, and sent out only at a release or a synchronization (depending on whether release consistency or weak ordering is being followed) or when the buffer becomes full. Or they may be buffered at the destination, and only applied at the next acquire point by that destination. This approach has been called delayed consistency [DWB+91] since it delays the propagation of invalidations. As long as processors do not see the invalidations, they continue to use their copies without any coherence actions, alleviating false sharing effects.

of which the page is fetched twice by P1 and P2, once at the read of y and once at the write of x). The extra messages problem is even more significant when an update-based protocol is used. These issues and alternatives are discussed further in Exercise 9.6.

The best known SVM systems tend to use a lazier form of release consistency, called *lazy release consistency* or LRC, which propagates and applies invalidations to a given processor not at the release that follows those writes but only at the next acquire by that processor [KCZ92]. On an acquire, the processor obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation. Identifying which are the release operations that occurred before the current acquire is an interesting question. One way to view this is to assume that this means all releases that would have to appear before this acquire in any sequentially consistent[1] ordering of the synchronization operations that preserves dependences. Another way of putting this is that there are two types of partial orders imposed on synchronization references: program order within each process, and dependence order among acquires and releases to the same synchronization variable. Accesses to a synchronization variable form a chain of successful acquires and releases. When an acquire comes to a releasing process P, the synchronization operations that occurred before that release are those that occur before it in the intersection of these program orders and dependence orders. These synchronization operations are said to have occurred before it in a *causal* sense. Figure 9-14 shows an example.

By further postponing coherence actions, LRC alleviates the three problems associated with ERC; for example, if false sharing on a page occurs before the acquire but not after it, its ill effects will not be seen (there is no page fault on the read of y in Figure 9-13(b)). On the other hand, some of the work to be done is shifted from release point to acquire point, and LRC is significantly more complex to implement than ERC as we shall see. However, LRC has been found to usually perform better, and is currently the method of choice.

The relationship between these software protocols and consistency models developed for hardware-coherent systems is interesting. First, the software protocols do not satisfy the requirements of coherence that were discussed in Chapter 5, since writes are not automatically guaranteed to be propagated unless the appropriate synchronization is present. Making writes visible only through synchronization operations also makes write serialization more difficult to guarantee— since different processes may see the same writes through different synchronization chains and hence in different orders—and most software systems do not provide this guarantee. The difference between hardware implementations of release consistency and ERC is in *when* writes are propagated; however, by propagating write notices only at acquires LRC may differ from release consistency even in *whether* writes are propagated, and hence may allow results that are not permitted under release conssistency. LRC is therefore a different consistency model than release consistency, while ERC is simply a different implementation of RC, and it requires greater programming care. However, if a program is properly labeled then it is guaranteed to run "correctly" under both RC and LRC.

---

1. Or even processor consistent, since RC allows acquires (reads) to bypass previous releases (writes).
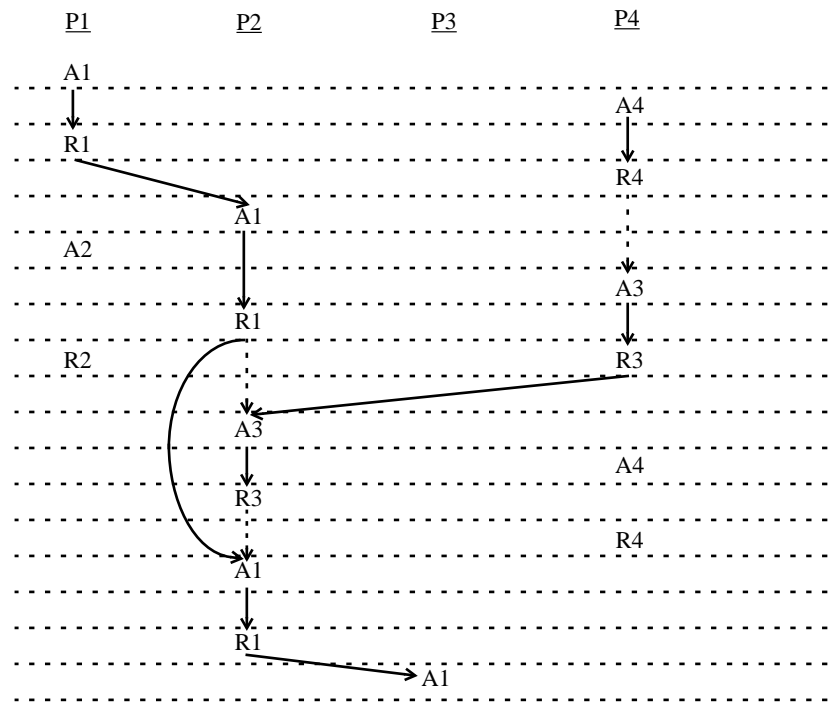
Figure 9-14 The causal order among synchronization events (and hence data accesses).

The figure shows what synchronization operations are before the acquire A1 (by process P3) or the release R1 by process P2 that enables it. The dotted horizontal lines are time-steps, increasing downward, indicating a possible interleaving in time. The bold arrows show dependence orders along an acquire-release chain, while the dotted arrows show the program orders that are part of the causal order. The A2, R2 and A4, R4 pairs that are untouched by arrows do not "happen before" the acquire of interest in causal order, so the accesses after the acquire are not guaranteed to see the accesses between them.

Example 9-2   Design an example in which LRC produces a different result than release consistency? How would you avoid the problem?

Answer   Consider the code fragment below, assuming the pointer ptr is initialized to NULL.

<u>P1</u>   <u>P2</u>

                                                            while (ptr == null) { };
lock L1;   **lock L1**;
ptr = non_null_ptr_val;   a = ptr;
 unlock L1;   **unlock L1**;

Under RC and ERC, the new non-null pointer value is guaranteed to propagate to P2 before the unlock (release) by P1 is complete, so P2 will see the new value and jump out of the loop as expected. Under LRC, P2 will not see the write by P1 until it performs its lock (acquire operation); it will therefore enter the while loop, never

see the write by P1, and hence never exit the while loop. The solution is to put the appropriate acquire synchronization before the reads in the while loop, or to label the reads of ptr appropriately as acquire synchronizations.

The fact that coherence information is propagated only at synchronization operations *that are recognized by the software SVM layer* has an interesting, related implication. It may be difficult to run existing application binaries on relaxed SVM systems, even if those binaries were compiled for systems that support a very relaxed consistency model and they are properly labeled. The reason is that the labels are compiled down to the specific fence instructions used by the commercial microprocessor, and those fence instructions may not be visible to the software SVM layer unless the binaries are instrumented to make them so. Of course, if the source code with the labels is available, then the labels can be translated to primitives recognized by the SVM layer.

Multiple Writer Protocols

Relaxing the consistency model works very well in mitigating the effects of false sharing when only one of the sharers writes the page in the interval between two synchronization points, as in our previous examples. However, it does not in itself solve the *multiple-writer* problem. Consider the revised example in Figure 9-15. Now P0 and P1 both modify the same page between the same
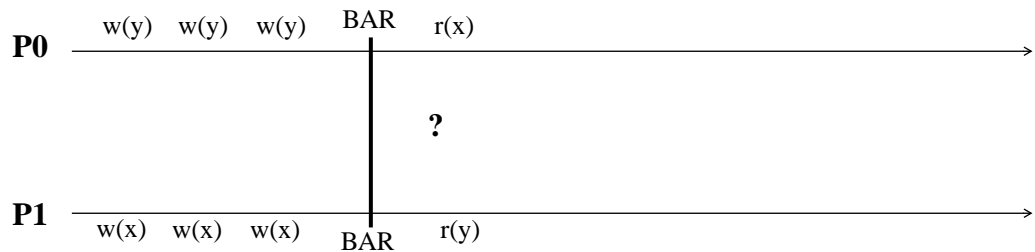
Figure 9-15 The multiple-writer problem.

At the barrier, two different processors have written the same page independently, and their modifications need to be merged.

two barriers. If we follow a single-writer protocol, as in hardware cache coherence schemes, then each of the writers must obtain ownership of the page before writing it, leading to ping-ponging communication even between the synchronization points and compromising the goal of using relaxed consistency in SVM. In addition to relaxed consistency, we need a "multiple-writer" protocol. This is a protocol that allows each processor writing a page between synchronization points to modify its own copy locally, allowing the copies to become inconsistent, and makes the copies consistent only at the next synchronization point, as needed by the consistency model and implementation (lazy or eager). Let us look briefly at some multiple writer mechanisms that can be used with either eager or lazy release consistency.

The first method is used in the TreadMarks SVM system from Rice University [KCD+94]. The idea is quite simple. To capture the modifications to a shared page, it is initially write-protected. At the first write after a synchronization point, a protection violation occurs. At this point, the system makes a copy of the page (called a twin) in software, and then unprotects the actual page so further writes can happen without protection violations. Later, at the next release or incoming

acquire (for ERC or LRC), the twin and the current copy are compared to create a "diff", which is simply a compact encoded representation of the differences between the two. The diff therefore captures the modifications that processor has made to the page in that synchronization interval. When a processor incurs a page fault, it must obtain the diffs for that page from other processors that have created them, and merge them into its copy of the page. As with write notices, there are several alternatives for when we might compute the diffs and when we might propagate them to other processors with copies of the page (see Exercise 9.6). If diffs are propagated eagerly at a release, they and the corresponding write notices can be freed immediately and the storage reused. In a lazy implementation, diffs and write notices may be kept at the creator until they are requested. In that case, they must be retained until it is clear that no other processor needs them. Since the amount of storage needed by these diffs and write notices can become very large, garbage collection becomes necessary. This garbage collection algorithm is quite complex and expensive, since when it is invoked each page may have uncollected diffs distributed among many nodes [KCD+94].

An alternative multiple writer method gets around the garbage collection problem for diffs while still implementing LRC, and makes a different set of performance tradeoffs [ISL96b,ZIL96]. The idea here is to not maintain the diffs at the writer until they are requested nor to propagate them to all the copies at a release. Instead, every page has a home memory, just like in flat hardware cache coherence schemes, and the diffs are propagated to the home at a release. The releasing processor can then free the storage for the diffs as soon as it has sent them to the home. The arriving diffs are merged into the home copy of the page (and the diff storage freed there too), which is therefore kept up to date. A processor performing an acquire obtains write notices for pages just as before. However, when it has a subsequent page fault on one of those pages, it does not obtain diffs from all previous writers but rather fetches the whole page from the home. Other than storage overhead and scalability, this scheme has the advantage that on a page fault only one round-trip message is required to fetch the data, whereas in the previous scheme diffs had to be obtained from all the previous ("multiple") writers. Also, a processor never incurs a page fault for a page for which it is the home. The disadvantages are that whole pages are fetched rather than diffs (though this can be traded off with storage and protocol processing overhead by storing the diffs at the home and not applying them there, and then fetching diffs from the home), and that the distribution of pages among homes becomes important for performance despite replication in main memory. Which scheme performs better will depend on how the application sharing patterns manifest themselves at page granularity, and on the performance characteristics of the communication architecture.

Alternatives Methods for Propagating Writes

Diff processing—twin creation, diff computation, and diff application—incurs significant overhead, requires substantial additional storage, and can also pollute the first-level processor cache, replacing useful application data. Some recent systems provide hardware support for fine-grained communication in the network interface—particularly fine-grained propagation of writes to remote memories—which that can be used to accelerate these home-based, multiple-writer SVM protocols and avoid diffs altogether. As discussed in Chapter 7, the idea of hardware propagation of writes originated in the PRAM [LS88] and PLUS [BR90] systems, and modern examples include the network interface of the SHRIMP multicomputer prototype at Princeton University [BLA+94] and the Memory Channel from Digital Equipment Corporation [GCP96].

For example, the SHRIMP network interface allows unidirectional mappings to be established between a pair of pages on different nodes, so that the writes performed to the source page are automatically snooped and propagated in hardware to the destination page on the other node. To detect these writes, a portion of the "assist" snoops the memory bus. However, it does not have to arbitrate for the memory bus, and the rest of it sits on the I/O bus, so the hardware is not very intrusive in the node. Of course, this "automatic update" mechanism requires that the cache be write-through, or that writes to remotely mapped pages somehow do appear on the memory bus at or before a release. This can be used to construct a home-based multiple writer scheme without diffs as follows [IDF+96,ISL96a]. Whenever a processor asks for a copy of the page, the system establishes a mapping from that copy to the home version of the page (see Figure 9-16). Then,

(a) The automatic update mechanism   (b) Supporting multiple writers using automatic update
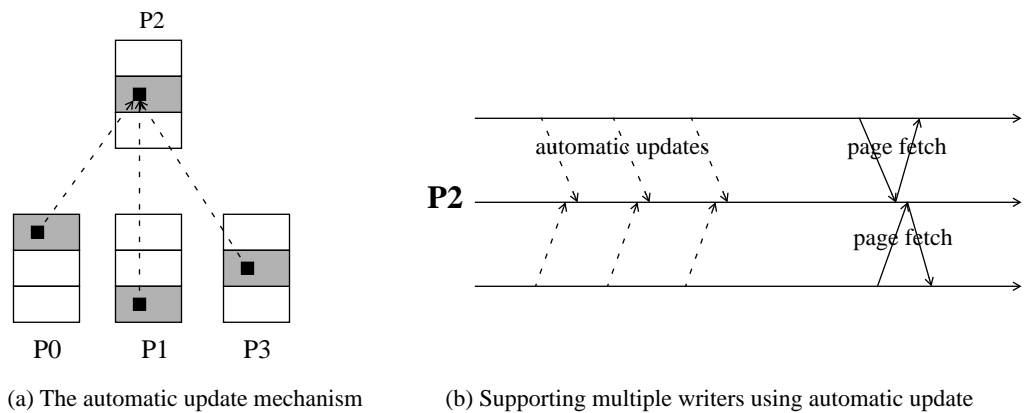
Figure 9-16 Using an automatic update mechanism to solve the multiple writer problem.

The variables x and y fall on the same page, which has node P2 as its home. If P0 (or P1) were the home, it would not need to propagate automatic updates and would not incur page faults on that page (only the other node would).

writes issued by the non-home processor to the page are automatically propagated by hardware to the home copy, which is thus always kept up to date. The outgoing link that carries modifications to their homes is flushed at a release point to ensure that all updates will reach (point to point network order is assumed). If a processor receives write notices for a page on an acquire (coherence actions are managed at synchronization points exactly as before), then when it faults on that page it will fetch the entire page from the home.

The DEC Memory Channel interface provides a similar mechanism, but it does not snoop the memory bus. Instead, special I/O writes have to be performed to mapped pages and these writes are automatically propagated to the mapped counterparts in main memory at the other end. Compared to the all-software home-based scheme, these hardware "automatic update" schemes have the advantage of not requiring the creation, processing and application of diffs. On the other hand, they increase data traffic, both on the memory bus when shared pages are cached write-through and in the network: Instead of communicating only the final diff of a page at a release, every write to a page whose home is remote is propagated to the home in fine-grained packets even between synchronization points, so multiple writes to the same location will all be propagated. The other disadvantage, of course, is that it uses hardware support, no matter how non-intrusive.

Even without this hardware support, twinning and diffing are not the only solutions for the multiple writers problem even in all software scheme. What we basically need to convey is what has

changed on that page during that interval. An alternative mechanism is to use *software dirty bits*. In this case, a bit is maintained for every word (or block) in main memory that keeps track of whether that block has been modified. This bit is set whenever that word is written, and the words whose bits are set at a release or acquire point constitute the diff equivalent that needs to be communicated. The problem is that the dirty bits must be set and reset in separate instructions from the stores themselves—since microprocessors have no support for this—so the program must be instrumented to add an instruction to set the corresponding bit at every shared store. The instrumentation is much like that needed for the all-software fine-grained coherence schemes mentioned earlier, and similar optimizations can be used to eliminate redundant setting of dirty bits. The advantage of this approach is the elimination of twinning and diff creation; the disadvantage is the extra instructions executed with high frequency.

The discussion so far has focussed on the functionality of different degrees of laziness, but has not addressed implementation. How do we ensure that the necessary write notices to satisfy the partial orders of causality get to the right places at the right time, and how do we reduce the number of write notices transferred. There is a range of methods and mechanisms for implementing release consistent protocols of different forms (single versus multiple writer, acquire- versus release-based, degree of laziness actually implemented). The mechanisms, what forms of laziness each can support, and their tradeoffs are interesting, and will be discussed in the Advanced Topics in Section 9.7.2.

Summary: Path of a Read Operation

To summarize the behavior of an SVM system, let us look at the path of a read. We examine the behavior of a home-based system since it is simpler. A read reference first undergoes address translation from virtual to physical address in the processor's memory management unit. If a local page mapping is found, the cache hierarchy is looked up with the physical address and it behaves just like a regular uniprocessor operation (the cache lookup may of course be done in parallel for a virtually indexed cache). If a local page mapping is not found, a page fault occurs and then the page is mapped in (either from disk or from another node), providing a physical address. If the operating system indicates that the page is not currently mapped on any other node, then it is mapped in from disk with read/write permission, otherwise it is obtained from another node with read-only permission. Now the cache is looked up. If inclusion is preserved between the local memory and the cache, the reference will miss and the block is loaded in from local main memory where the page is now mapped. Note that inclusion means that a page must be flushed from the cache (or the state of the cache blocks changed) when it is invalidated or downgraded from read/write to read-only mode in main memory. If a write reference is made to a read-only page, then also a page fault is incurred and ownership of the page obtained before the reference can be satisfied by the cache hierarchy.

Performance Implications

Lazy release consistency and multiple writer protocols improve the performance of SVM systems dramatically compared to sequentially consistent implementations. However, there are still many performance problems compared with machines that manage coherence in hardware at cache block granularity. The problems of false sharing and either extra communication or protocol processing overhead do not disappear with relaxed models, and the page faults that remain are still expensive to satisfy. The high cost of communication, and the contention induced by higher end-point processing overhead, magnifies imbalances in communication volume and

hence time among processors. Another problem in SVM systems is that synchronization is performed in software through explicit software messages, and is very expensive. This is exacerbated by the fact that page misses often occur within critical sections, artificially dilate them and hence greatly increasing the serialization at critical sections. The result is that while applications with coarse-grained data sharing patterns (little false sharing and communication fragmentation) perform quite well on SVM systems, applications with finer-grained sharing patterns do not migrate well from hardware cache-coherent systems to SVM systems. The scalability of SVM systems is also undetermined, both in performance and in the ability to run large problems since the storage overheads of auxiliary data structures grow with the number of processors.

All in all, it is still unclear whether fine-grained applications that run well on hardware-coherent machines can be restructured to run efficiently on SVM systems as well, and whether or not such systems are viable for a wide range of applications. Research is being done to understand the performance issues and bottlenecks [DKC+93, ISL96a, KHS+97, JSS97], as well as the value of adding some hardware support for fine-grained communication while still maintaining coherence in software at page granularity to strike a good balance between hardware and software support [KoS96, ISL96a, BIS97]. With the dramatically increasing popularity of low-cost SMPs, there is also a lot of research in extending SVM protocols to build coherent shared address space machines as a two-level hierarchy: hardware coherence within the SMP nodes, and software SVM coherence across SMP nodes [ENC+96, SDH+97, SBI+97]. The goal of the outer SVM protocol is to be invoked as infrequently as possible, and only when cross-node coherence is needed.

9.4.4 Access Control through Language and Compiler Support

Language and compiler support can also be enlisted to support coherent replication. One approach is to program in terms of data objects of "regions" of data, and have the runtime system that manages these objects provide access control and coherent replication at the granularity of objects or regions. This "shared object space" programming model motivates the use of even more relaxed memory consistency models, as we shall see next. We shall also briefly discuss compiler-based coherence and approaches that provide a shared address space in software but do not provide automatic replication and coherence.

Object-based Coherence

The release consistency model takes advantage of the *when* dimension of memory consistency. That is, it tells us by when it is necessary for writes by one process to be performed with respect to another process. This allows successively lazy implementations to be developed, as we have discussed, which delay the performing of writes as long as possible. However, even with release consistency, if the synchronization in the program requires that process P1's writes be performed or become visible at process P2 by a certain point, then this means that *all* of P1's writes to all the data it wrote must become visible (invalidations or updates conveyed), even if P2 does not need to see all the data. More relaxed consistency models take into the account the *what* dimension, by propagating invalidations or updates only for those data that the process acquiring the synchronization may need. The when dimension in release consistency is specified by the programmer through the synchronization inserted in the program. The question is how to specify the what dimension. It is possible to associate with a synchronization event the set of pages that must be made consistent with respect to that event. However, this is very awkward for a programmer to do.

write A

acquire (L)

write B

release (L) _ _ _ _ _ _ → acquire (L)
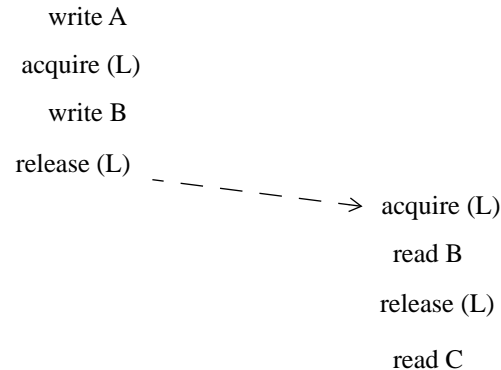
read B

release (L)

read C

Figure 9-17 Why release consistency is conservative.

Suppose A and B are on different pages. Since P1 wrote A and B before releasing L (even though A was not written inside the critical section), invalidations for both A and B will be propagated to P2 and applied to the copies of A and B there. However, P2 does not need to see the write to A, but only that to B. Suppose now P2 reads another variable C that resides on the same page as A. Since the page containing A has been invalidated, P2 will incur a page miss on its access to C due to false sharing. If we could somehow associate the page containing B with the lock L1, then only the invalidation of B can be propagated on the acquire of L by P2, and the false sharing miss would be saved.

Region- or object-based approaches provide a better solution. The programmer breaks up the data into logical objects or regions (regions are arbitrary, user-specified ranges of virtual addresses that are treated like objects, but do not require object-oriented programming). A runtime library then maintains consistency at the granularity of these regions or objects, rather than leaving it entirely to the operating system to do at the granularity of pages. The disadvantages of this approach are the additional programming burden of specifying regions or objects appropriately, and the need for a sophisticated runtime system between the application and the OS. The major advantages are: (i) the use of logical objects as coherence units can help reduce false sharing and fragmentation to begin with, and (ii) they provide a handle on specifying data logically, and can be used to relax the consistency using the what dimension.

For example, in the *entry consistency* model [BZS93], the programmer associates a set of data (regions or objects) with every synchronization variable such as a lock or barrier (these associations or bindings can be changed at runtime, with some cost). At synchronization events, only those objects or regions that are associated with that synchronization variable are guaranteed to be made consistent. Write notices for other modified data do not have to be propagated or applied. An alternative is to specify the binding for each synchronization event rather than variable. If no bindings are specified for a synchronization variable, the default release consistency model is used. However, the bindings are not hints: If they are specified, they must be complete and correct, otherwise the program will obtain the wrong answer. The need for explicit, correct bindings imposes a substantial burden on the programmer, and sufficient performance benefits have not yet been demonstrated to make this worthwhile. The Jade language achieves a similar effect, though by specifying data usage in a different way [RSL93]. Finally, attempts have been made to exploit the association between synchronization and data implicitly in a page-based shared virtual memory approach, using a model called *scope consistency* [ISL96b].

Compiler-based Coherence

Research has been done in having the compiler keep caches coherent in a shared address space, using some additional hardware support in the processor system. These approaches rely on the compiler (or programmer) to identify parallel loops. A simple approach to coherence is to insert a barrier at the end of every parallel loop, and flush the caches between loops. However, this does not allow any data locality to be exploited in caches across loops, even for data that are not actively shared at all. More sophisticated approaches have been proposed that require support for selective invalidations initiated by the processor and fairly sophisticated hardware support to keep track of version numbers for cache blocks. We shall not describe these here, but the interested reader can refer to the literature [CV90]. Other than non-standard hardware and compiler support for coherence, the major problem with these approaches is that they rely on the automatic parallelization of sequential programs by the compiler, which is simply not there yet for realistic programs.

Shared Address Space without Coherent Replication

Systems in this category supports a shared address space abstraction through the language and compiler, but without automatic replication and coherence, just like the Cray T3D and T3E did in hardware. One type of example is a data parallel languages (see Chapter 2) like High Performance Fortran. The distributions of data specified by the user, together with the owner computes rule, are used by the compiler or runtime system to translate off-node memory references to explicit send/receive messages, to make messages larger, or to align data for better spatial locality etc. Replication and coherence are usually up to the user, which compromises ease of programming, or system software may try to replicate in main memory automatically. Efforts similar to HPF are being made with languages based on C and C++ as well [BBG+93,LRV96].

A more flexible language and compiler based approach is taken by the Split-C language [CDG+93]. Here, the user explicitly specifies arrays as being local or global (shared), and for global arrays specifies how they should be laid out among physical memories. Computation may be assigned independently of the data layout, and references to global arrays are converted into messages by the compiler or runtime system based on the layout. The decoupling of computation assignment from data distribution makes the language much more flexible than an owner computes rule for load balancing irregular programs, but it still does not provide automatic support for replication and coherence which can be difficult to manage. Of course, all these software systems can be easily ported to hardware-coherent shared address space machines, in which case the shared address space, replication and coherence are implicitly provided.

9.5 Putting it All Together: A Taxonomy and Simple-COMA

The approaches to managing replication and coherence in the extended memory hierarchy discussed in this chapter have had different goals: improving performance by replicating in main memory, in the case of COMA, and reducing cost, in the case of SVM and the other systems of the previous section. However, it is useful to examine the management of replication and coherence in a unified framework, since this leads to the design of alternative systems that can pick and choose aspects of existing ones. A useful framework is one that distinguishes the approaches along two closely related axes: the granularities at which they allocate data in the replication store, keep data coherent and communicate data between nodes, and the degree to which they uti-

lize additional hardware support in the communication assist beyond that available in uniprocessor systems. The two axes are related because some functions are either not possible at fine granularity without additional hardware support (e.g. allocation of data in main memory) or not possible with high performance. The framework applies whether replication is done only in the cache (as in CC-NUMA) or in main memory as well, though we shall focus on replication in main memory for simplicity.

Figure 9-18 depicts the overall framework and places different types of systems in it. We divide
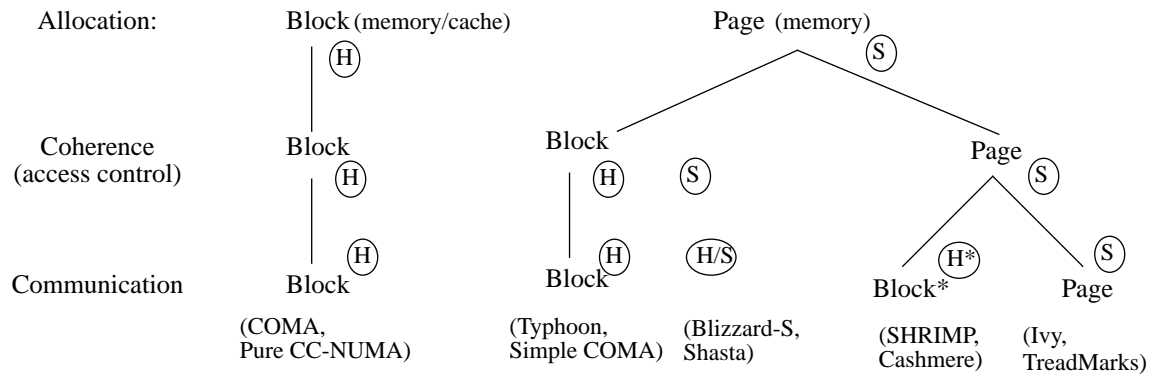

Figure 9-18 Granularities of allocation, coherence and communication in coherent shared address space systems.

The granularities specified are for the replication store in which data are first replicated automatically when they are brought into a node. This level is main memory in all the systems, except for pure CC-NUMA where it is the cache. Below each leaf in the taxonomy are listed some representative systems of that type, and next to each node is a letter indicating whether the support for that function is usually provided in hardware (H) or software (S). The asterisk next to block and H in one case means that not all communication is performed at fine granularity. Citations for these systems include: COMA [HLH+92, SJG92, FBR93], Typhoon [RLW94], Simple COMA [SWC+95], Blizzard-S [SFL+94], Shasta [LGT96], SHRIMP [BLA+94,IDF+96], Cashmere [KoS96], Ivy [LiH89], TreadMarks [KCD+94].

granularities into "page" and "block" (cache block), since these are the most common in systems that do not require a stylized programming model such as objects. However, "block" includes other fine granularities such as individual words, and "page" also covers coarse-grained objects or regions of memory that may be managed by a runtime system.

On the left side of the figure are COMA systems, which allocate space for data in main memory at the granularity of cache blocks, using additional hardware support for tags etc. Given replication at cache block granularity, it makes sense to keep data coherent at a granularity at least this fine as well, and also to communicate at fine granularity.[1]

On the right side of the figure are systems that allocate and manage space in main memory at page granularity, with no extra hardware needed for this function. Such systems may provide

---

1. This does not mean that fine-grained allocation necessarily implies fine-grained coherence or communication. For example, is possible to exploit communication at coarse grain even when allocation and coherence are at fine grain, to gain the benefits of large data transfers. However, the situations discussed are indeed the common case, and we shall focus on these.

access control and hence coherence at page granularity as well, as in SVM, in which case they may either provide or not provide support for fine-grained communication as we have seen. Or they may provide access control and coherence at block granularity as well, using either software instrumentation and per-block tags and state or hardware support for these functions. Systems that support coherence at fine granularity typically provide some form of hardware support for efficient fine-grained communication as well.

9.5.1 Putting it All Together: Simple-COMA and Stache

While COMA and SVM both replicate in main memory, and each addresses some but not all of the limitations of CC-NUMA, they are at two ends of the spectrum in the above taxonomy. While COMA is a hardware intensive solution and maintains fine granularities, issues in managing main memory such as the last copy problem are challenging for hardware. SVM leaves these complex memory management problems to system software—which also allows main memory to be a fully associative cache through the OS virtual to physical mappings—but its performance may suffer due to the large granularities and software overheads of coherence. The framework in Figure 9-18 leads to interesting ways to combine the cost and hardware simplicity of SVM with the performance advantages and ease of programming of COMA; namely the Simple-COMA [SWC+95] and Stache [RLW94] approaches shown in the middle of the figure. These approaches divide the task of coherent replication in main memory into two parts: (i) memory management (address translation, allocation and replacement), and (ii) coherence, including replication and communication and coherence. Like COMA, they provide coherence at fine granularity with specialized hardware support for high performance, but like SVM (and unlike COMA) they leave memory management to the operating system at page granularity. Let us begin with Simple COMA.

The major appeal of Simple COMA relative to COMA is design simplicity. Performing allocation and memory management through the virtual memory system instead of hardware simplifies the hardware protocol and allows fully associative management of the "attraction memory" with arbitrary replacement policies. To provide coherence or access control at fine grain in hardware, each page in a node's memory is divided into coherence blocks of any chosen size (say a cache block), and state information is maintained in hardware for each of these units. Unlike in COMA there is no need for tags since the presence check is done at page level. The state of the block is checked in parallel with the memory access for that unit when a miss occurs in the hardware cache. Thus, there are two levels of access control: for the page, under operating system control, and if that succeeds then for the block, under hardware control.

Consider the performance tradeoffs relative to COMA. Simple COMA reduces the latency in the path to access the local main memory (which is hopefully the frequent case among cache misses). There is no need for the hardware tag comparison and selection used in COMA, or in fact for the local/remote address check that is needed in pure CC-NUMA machines to determine whether to look up the local memory. On the other hand, every cache miss looks up the local memory (cache), so like in COMA the path of a nonlocal access is longer. Since a shared page can reside in different physical addresses in different memories, controlled independently by the node operating systems, unlike in COMA we cannot simply send a block's physical address across the network on a miss and use it at the other end. This means that we must support a shared virtual address space. However, the virtual address issued by the processor is no longer available by the time the attraction memory miss is detected. This requires that the physical address, which is available, be "reverse translated" to a virtual address or other globally consis-

tent identifier; this identifier is sent across the network and is translated back to a physical address by the other node. This process incurs some added latency, and will be discussed further in Section 9.7.

Another drawback of Simple COMA compared to COMA is that although communication and coherence are at fine granularity, allocation is at page grain. This can lead to fragmentation in main memory when the access patterns of an application do not match page granularity well. For example, if a processor accesses only one word of a remote page, only that coherence block will be communicated but space will be allocated in the local main memory for the whole page. Similarly, if only one word is brought in for an unallocated page, it may have to replace an entire page of useful data (fortunately, the replacement is fully associative and under the control of software, which can make sophisticated choices). In contrast, COMA systems typically allocate space for only that coherence block. Simple COMA is therefore more sensitive to spatial locality than COMA.

An approach similar to Simple COMA is taken in the Stache design proposed for the Typhoon system [SWC+95] and implemented in the Typhoon-zero research prototype [RPW96]. Stache uses the tertiary cache approach discussed earlier for replication in main memory, but with allocation managed at page level in software and coherence at fine grain in hardware. Other differences from Simple COMA are that the assist in the Typhoon systems is programmable, physical addresses are reverse translated to virtual addresses rather than other global identifiers (to enable user-level software protocol handlers, see Section 9.7). Designs have also been proposed to combine the benefits of CC-NUMA and Simple COMA [FaW97].

Summary: Path of a Read Reference

Consider the path of a read in Simple COMA. The virtual address is first translated to a physical address by the processor's memory management unit. If a page fault occurs, space must be allocated for a new page, though data for the page are not loaded in. The virtual memory system decides which page to replace, if any, and establishes the new mapping. To preserve inclusion, data for the replaced page must be flushed or invalidated from the cache. All blocks on the page are set to invalid. The physical address is then used to look up the cache hierarchy. If it hits (it will not if a page fault occurred), the reference is satisfied. If not, then it looks up the local attraction memory, where by now the locations are guaranteed to correspond to that page. If the block of interest is in a valid state, the reference completes. If not, the physical address is "reverse translated" to a global identifier, which plays the role of a virtual address, which is sent across the network guided by the directory coherence protocol. The remote node translates this global identifier to a local physical address, uses this physical address to find the block in its memory hierarchy, and sends the block back to the requestor. The data is then loaded into the local attraction memory and cache, and delivered to the processor.

To summarize the approaches we have discussed, Figure 9-19 summarizes the node structure of the approaches that use hardware support to preserve coherence at fine granularity.

9.6 Implications for Parallel Software

Let us examine the implications for parallel software of all the approaches discussed in this chapter, beyond those discussed already in earlier chapters.
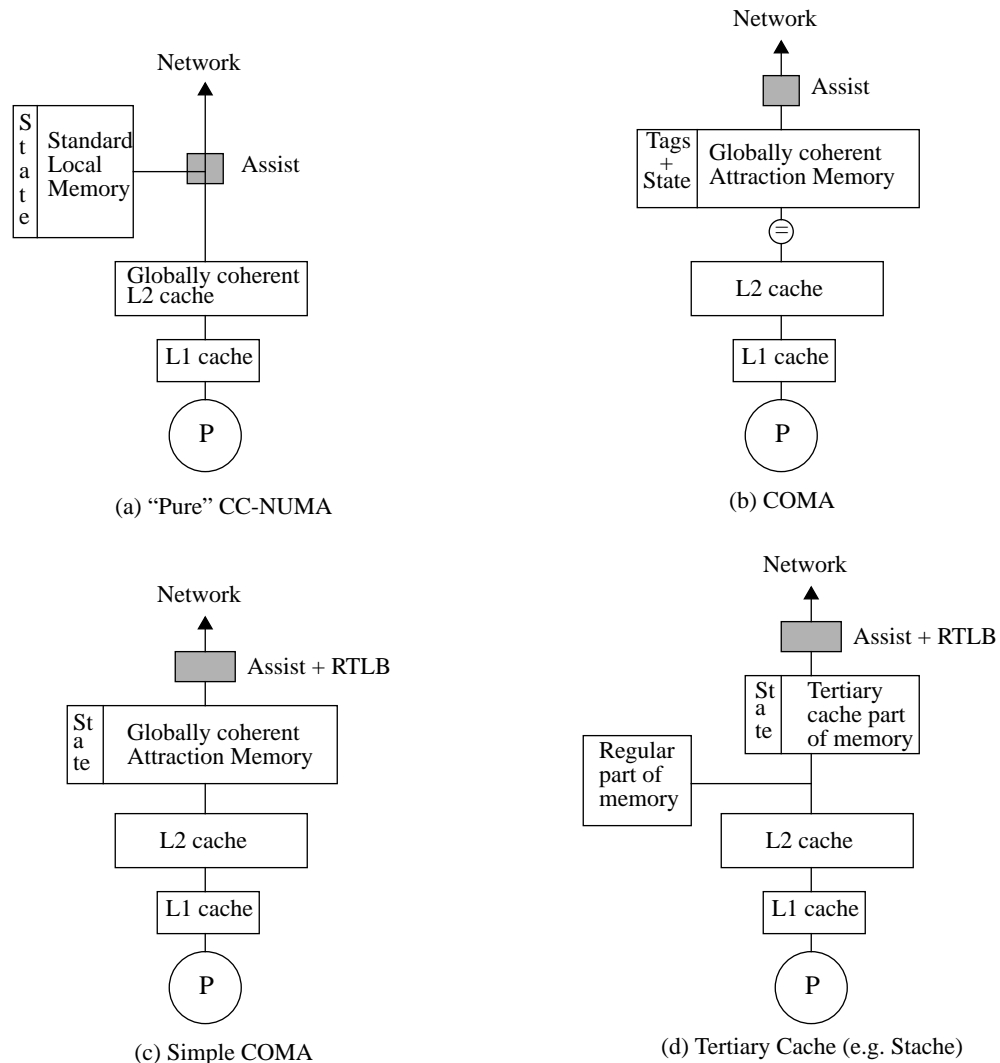
Figure 9-19 Logical structure of a typical node in four approaches to a coherent shared address space. A vertical path through the node in the figures traces the path of a read miss that must be satisfied remotely (going through main memory means that main memory must be looked up, though this may be done in parallel with issuing the remote request if speculative lookups are used).

Relaxed memory consistency models require that parallel programs label the desired conflicting accesses as synchronization points. This is usually quite stylized, for example looking for variables that a process spins on in a while loop before proceeding, but sometimes orders must be preserved even without spin-waiting as in some of the examples in Figure Figure 9-4. A programming language may provide support to label some variables or accesses as synchronization, which will then be translated by the compiler to the appropriate order-preserving instruction. Current programming languages do not provide integrated support for such labeling, but rely on the programmer inserting special instructions or calls to a synchronization library. Labels can also be used by the compiler itself, to restrict its own reorderings of accesses to shared memory.

One mechanism is to declare some (or all) shared variables to be of a "volatile" type, which means that those variables will not be allocated in registers and will not be reordered with respect to the accesses around them. Some new compilers also recognize explicit synchronization calls and disallow reorderings across them (perhaps distinguishing between acquire and release calls), or obey orders with respect to special order-preserving instructions that the program may insert.

The automatic, fine-grained replication and migration provided by COMA machines is designed to allows the programmer to ignore the distributed nature of main memory in the machine. It is very useful when capacity or conflict misses dominate. Experience with parallel applications indicates that because of the sizes of working sets and of caches in modern systems, the migration feature may be more broadly useful than the replication and coherence, although systems with small caches or applications with large, unstructured working sets can benefit from the latter as well. Fine-grained, automatic migration is particularly useful when the data structures in the program cannot easily be distributed appropriately at page granularity, so page migration techniques such as those provided by the Origin2000 may not be so successful; for example, when data that should be allocated on two different nodes fall on the same page (see Section 8.10). Data migration is also very useful in multiprogrammed workloads, consisting of even sequential programs, in which application processes are migrated among processing nodes by the operating system for load balancing. Migrating a process will turn what should be local misses into remote misses, unless the system moves all the migrated process's data to the new node's main memory as well. However, this case is likely to be handled quite well by software page migration.

In general, while COMA systems suffer from higher communication latencies than CC-NUMA systems, they may allow a wider class of applications to perform well with less programming effort. However, explicit migration and proper data placement can still be moderately useful even on these systems. This is because ownership requests on writes still go to the directory, which may be remote and does not migrate with the data, and thus can cause extra traffic and contention even if only a single processor ever writes a page.

Commodity-oriented systems put more pressure on software not only to reduce communication volume, but also to orchestrate data access and communication carefully since the costs and/or granularities of communication are much larger. Consider shared virtual memory, which performs communication and coherence at page granularity. The actual memory access and data sharing patterns interact with this granularity to produce an induced sharing pattern at page granularity, which is the pattern relevant to the system [ISL96a]. Other than by high communication to computation ratio, performance is adversely affected when the induced pattern involves write-sharing and hence multiple writers of the same page, or fine grained access patterns in which references from different processes are interleaved at a fine granularity. This makes it important to try to structure data accesses so that accesses from different processes tend not to be interleaved at a fine granularity in the address space. The high cost of synchronization and the dilation of critical sections makes it important to reduce the use of synchronization in programming for SVM systems. Finally, the high cost of communication and synchronization may make it more difficult to use task stealing successfully for dynamic load balancing in SVM systems. The remote accesses and synchronization needed for stealing may be so expensive that there is little work left to steal by the time stealing is successful. It is therefore much more important to have a well balanced initial assignment of tasks in a task stealing computation in an SVM system than in a hardware-coherent system. In similar ways, the importance of programming and algorithmic optimizations depend on the communication costs and granularities of the system at hand.

9.7 Advanced Topics

Before we conclude this chapter, let us discuss two other limitations of the traditional CC-NUMA approach, and examine (as promised) the mechanisms and techniques for taking advantage of relaxed memory consistency models in software, for example for shared virtual memory protocols.

9.7.1 Flexibility and Address Constraints in CC-NUMA Systems

The two other limitations of traditional CC-NUMA systems mentioned in the introduction to this chapter were the fact that a single coherence protocol is hard-wired into the machine and the potential limitations of addressability in a shared physical address space. Let us discuss each in turn.

Providing Flexibility

One size never really fits all. It will always be possible to find workloads that would be better served by a different protocol than the one hard-wired into a given machine. For example, while we have seen that invalidation-based protocols overall have advantages over update-based protocols for cache-coherent systems, update-based protocols are advantageous for purely producer-consumer sharing patterns. For a one-word producer-consumer interaction, in an invalidation-based protocol the producer will generate an invalidation which will be acknowledged, and then the consumer will issue a read miss which the producer will satisfy, leading to four network transactions; an update-based protocol will need only one transaction in this case. As another example, if large amounts of pre-determinable data are to be communicated from one node to another, then it may be useful to transfer them in a single explicit large message than one cache block at a time through load and store misses. A single protocol does not even necessarily best fit all phases or all data structures of a single application, or even the same data structure in different phases of the application. If the performance advantages of using different protocols in different situations are substantial, it may be useful to support multiple protocols in the communication architecture. This is particularly likely when the penalty for mismatched protocols is very high, as in commodity-based systems with less efficient communication architectures.

Protocols can be altered—or mixed and matched—by making the protocol processing part of the communication assist programmable rather than hard-wired, thus implementing the protocol in software rather than hardware. This is clearly quite natural in page-based coherence protocols where the protocol is in the software page fault handlers. For cache-block coherence it turns out that the requirements placed on a programmable controller by different protocols are usually very similar. On the control side, they all need quick dispatch to a protocol handler based on a transaction type, and support for efficient bit-field manipulation in tags. On the data side, they need high-bandwidth, low-overhead pipelined movement of data through the controller and network interface. The Sequent NUMA-Q discussed earlier provides a pipelined, specialized programmable coherence controller, as does the Stanford FLASH design [KOH+94, HKO+94]. Note that the controller being programmable doesn't alter the need for specialized hardware support for coherence; those issues remain the same as with a fixed protocol.

The protocol code that runs on the controllers in these fine-grained coherent machines operates in privileged mode so that it can use the physical addresses that it sees on the bus directly. Some

researchers also advocate that users be allowed to write their own protocol handlers in user-level software so they can customize protocols to exactly the needs of individual applications [FLR+94]. While this can be advantageous, particularly on machines with less efficient communication architectures, it introduces several complications. One cause of complications is similar to the issue discussed earlier for Simple COMA systems, though now for a different reason. Since the address translation is already done by the processor's memory management unit by the time a cache miss is detected, the assist sees only physical addresses on the bus. However, to maintain protection, user-level protocol software cannot be allowed access to these physical addresses. So if protocol software is to run on the assist at user-level, the physical addresses must be reverse translated back to virtual addresses before this software can use them. Such reverse translation requires further hardware support and increases latency, and it is complicated to implement. Also, since protocols have many subtle complexities related to correctness and deadlock and are difficult to debug, it is not clear how desirable it is to allow users to write protocols that may deadlock a shared machine.

Overcoming Physical Address Space Limitations

In a shared physical address space, the assist making a request sends the *physical* address of the location (or cache line) across the network, to be interpreted by the assist at the other end. As discussed for Simple COMA, this has the advantage that the assist at the other end does not have to translate addresses before accessing physical memory. However, a problem arises when the physical addresses generated by the processor may not have enough bits to serve as global addresses for the entire shared address space, as we saw for the Cray T3D in Chapter 7 (the Alpha 21064 processor emitted only 32 bits of physical address, insufficient to address the 128 GB or 37 bits of physical memory in a 2048-processor machine). In the T3D, segmentation through the annex registers was used to extend the address space, potentially introducing delays into the critical paths of memory accesses. The alternative is to not have a shared physical address space, but send virtual addresses across the network which will be retranslated to physical addresses at the other end.

SVM systems as well as Simple COMA systems use this approach of implementing a shared virtual rather than physical address space. One advantage of this approach is that now only the virtual addresses need to be large enough to index the entire shared (virtual) address space; physical addresses need only be large enough to address a given processor's main memory, which they certainly ought to be. A second advantage, seen earlier, is that each node manages its own address space and translations so more flexible allocation and replacement policies can be used in main memory. However, this approach does require address translation at both ends of each communication.

9.7.2 Implementing Relaxed Memory Consistency in Software

The discussion of shared virtual memory schemes that exploit release consistency showed that they can be either single-writer or multiple-writer, and can propagate coherence information at either release or acquire operations (eager and lazy coherence, respectively). As mentioned in that discussion, a range of techniques can be used to implement these schemes, successively adding complexity, enabling new schemes, and making the propagation of write notices lazier. The techniques are too complex and require too many structures to implement for hardware cache coherence, and the problems they alleviate are much less severe in that case due to the fine gran-

ularity of coherence, but they are quite well suited to software implementation. This section examines the techniques and their tradeoffs.

The basic questions for coherence protocols that were raised in Section 8.2 are answered somewhat differently in SVM schemes. To begin with, the protocol is invoked not only at data access faults as in the earlier hardware cache coherence schemes, but both at access faults (to find the necessary data) as well as at synchronization points (to communicate coherence information in write notices). The next two questions—finding the source of coherence information and determining with which processors to communicate—depend on whether release-based or acquire-based coherence is used. In the former, we need to send write notices to all valid copies at a release, so we need a mechanism to keep track of the copies. In the latter, the acquirer communicates with only the last releaser of the synchronization variable and pulls all the necessary write notices from there to only itself, so there is no need to explicitly keep track of all copies. The last standard question is communication with the necessary nodes (or copies), which is done with point-to-point messages.

Since coherence information is not propagated at individual write faults but only at synchronization events, a new question arises, which is how to determine for which pages write notices should be sent. In release-based schemes, at every release write notices are sent to all currently valid copies. This means that a node has only to send out write notices for writes that it performed since its previous release. All previous write notices in a causal sense have already been sent to all relevant copies, at the corresponding previous releases. In acquire-based methods, we must ensure that causally necessary write notices, which may have been produced by many different nodes, will be seen even though the acquirer goes only to the previous releaser to obtain them. The releaser cannot simply send the acquirer the write notices it has produced since its last release, or even the write notices it has produced ever. In both release- and acquire-based cases, several mechanisms are available to reduce the number of write notices communicated and applied. These include version numbers and time-stamps, and we shall see them as we go along.

To understand the issues more clearly, let us first examine how we might implement single-writer release consistency using both release-based and acquire-based approaches. Then, we will do the same thing for multiple-writer protocols.

Single Writer with Coherence at Release

The simplest way to maintain coherence is to send write notices at every release to all sharers of the pages that the releasing processor wrote since its last release. In a single-writer protocol, the copies can be kept track of by making the current owner of a page (the one with write privileges) maintain the current sharing list, and transferring the list at ownership changes (when another node writes the page). At a release, a node sends write notices for all pages it has written to the nodes indicated on its sharing lists (see Exercise 9.6).

There are two performance problems with this scheme. First, since ownership transfer does not cause copies to be invalidated (read-only copies may co-exist with one writable copy), a previous owner and the new owner may very well have the same nodes on their sharing lists. When both of them reach their releases (whether of the same synchronization variable or different ones) they will both send invalidations to some of the same pages, so a page may receive multiple (unnecessary) invalidations. This problem can be solved by using a single designated place to keep track of which copies of a page have already been invalidated, e.g. by using directories to keep track of

sharing lists instead of maintaining them at the owners. The directory is looked up before write notices are sent, and invalidated copies are recorded so multiple invalidations won't be sent to the same copy.

The second problem is that a release may invalidate a more recent copy of the page (but not the
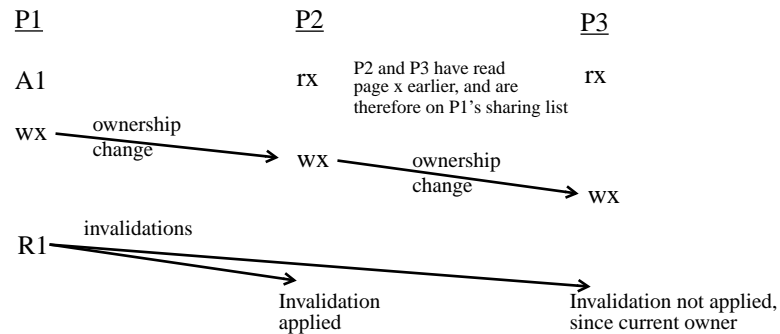

Figure 9-20 Invalidation of a more recent copy in a simple single-writer protocol.

Processor P2 steals ownership from P1, and then P3 from P2. But P1's release happens after all this. At the release, P1 sends invalidations to both P2 and P3. P2 applies the invalidation even though it's copy is more recent than P1's, since it doesn't know, while P3 does not apply the invalidation since it is the current owner.

most recent), which it needn't have done, as illustrated in Figure 9-20. This can be solved by associating a version number with each copy of a page. A node increments its version number for a page whenever it obtains ownership of that page from another node. Without directories, a processor will send write notices to all sharers (since it doesn't know their version numbers) together with its version number for that page, but only the receivers that have smaller version numbers will actually invalidate their pages. With directories, we can reduce the write notice traffic as well, not just the number of invalidations applied, by maintaining the version numbers at the directory entry for the page and only sending write notices to copies that have lower version numbers than the releaser. Both ownership and write notice requests come to the directory, so this is easy to manage. Using directories and version numbers together solves both the above problems. However, this is still a release-based scheme, so invalidations may be sent out and applied earlier than necessary for the consistency model, causing unnecessary page faults.

Single Writer with Coherence at Acquire

A simple way to use the fact that coherence activity is not needed until the acquire is to still have the releaser send out write notices to all copies, but do this not at the release but only when the next acquire request comes in. This delays the sending out of write notices; however, the acquire must wait until the releaser has sent out the write notices and acknowledgments have been received, which slows down acquire operation. The best bet for such fundamentally release-based approaches would be to send write notices out at the release, and wait for acknowledgments only before responding to the next incoming acquire request. This allows the propagation of write notices to be overlapped with the computation done between the release and the next incoming acquire.

Consider now the lazier, pull-based method of propagating coherence operations at an acquire, from the releaser to only that acquirer. The acquirer sends a request to the last releaser (or current

holder) of the synchronization variable, whose identity it can obtain from a designated manager node for that variable. This is the only place from where the acquirer is obtaining information, and it must see all writes that have happened before it in the causal order. With no additional support, the releaser must send to the acquirer all write notices that the releaser has either produced so far or received from others (at least since the last time it sent the acquirer write notices, if it keeps track of that) It cannot only send those that it has itself produced since the last release, since it has no idea how many of the necessary write notices the acquirer has already seen through previous acquires from another processors. The acquirer must keep those write notices to pass on to the next acquirer.

Carrying around an entire history of write notices is obviously not a good idea. Version numbers, incremented at changes of ownership as before, can help reduce the number of invalidations applied if they are communicated along with the write notices, but they do not help reduce the number of write notices sent. The acquirer cannot communicate version numbers to the releaser for this, since it has no idea what pages the releaser wants to send it write notices for. Directories with version numbers don't help, since the releaser would have to send the directory the history of write notices.

In fact, what the acquirer wants is the write notices corresponding to all releases that precede it causally and that it hasn't already obtained through its previous acquires. Keeping information at page level doesn't help, since neither acquirer nor releaser knows which pages the other has seen and neither wants to send all the information it knows. The solution is to establish a system of virtual time. Conceptually, every node keeps track of the virtual time period up to which it has seen write notices from each other node. An acquire sends the previous releaser this time vector; the releaser compares it with its own, and sends the acquirer write notices corresponding to time periods the releaser has seen but the acquirer hasn't. Since the partial orders to be satisfied for causality are based on synchronization events, associating time increments with synchronization events lets us represent these partial orders explicitly.

More precisely, the execution of every process is divided into a number of *intervals*, a new one beginning (and a local interval counter for that process incrementing) whenever the process successfully executes a release or an acquire. The intervals of different processes are partially ordered by the desired precedence relationships discussed earlier: (i) intervals on a single process are totally ordered by the program order, and (ii) an interval on process P precedes an interval on process Q if its release precedes Q's acquire for that interval in a chain of release-acquire operations on the same variable. However, interval numbers are maintained and incremented locally per process, so the partial order (ii) does not mean that the acquiring process's interval number will be larger than the releasing process's interval number. What it does mean, and what we are trying to ensure, is that if a releaser has seen write notices for interval 8 from process X, then the next acquirer should also have seen at least interval 8 from process X before it is allowed to complete its acquire. To keep track of what intervals a process has seen and hence preserve the partial orders, every process maintains a *vector timestamp* for each of its intervals [KCD+94]. Let V^P_i be the vector timestamp for interval i on process P. The number of elements in the vector V^P_i equals the number of processes. The entry for process P itself in V^P_i is equal to i. For any other process Q, the entry denotes the most recent interval of process Q that precedes interval i on process P in the partial orders. Thus, the vector timestamp indicates the most recent interval from every other process that this process ought to have already received and applied write notices for (through a previous acquire) by the time it enters interval i.

On an acquire, a process *P* needs to obtain from the last releaser *R* write notices pertaining to intervals (from any processor) that *R* has seen before its release but the acquirer has not seen through a previous acquire. This is enough to ensure causality: Any other intervals that *P* should have seen from other processes, it would have seen through previous acquires in its program order. *P* therefore sends its current vector timestamp to *R*, telling it which are the latest intervals from other processes it has seen before this acquire. *R* compares *P*'s incoming vector time-stamp with its own, entry by entry, and piggybacks on its reply to *P* write notices for all intervals that are in *R*'s current timestamp but not in *P*'s (this is conservative, since R's current time-stamp may have seen more than R had at the time of the relevant release). Since *P* has now received these write notices, it sets its new vector timestamp, for the interval starting with the current acquire, to the pairwise maximum of *R*'s vector timestamp and its own previous one. *P* is now up to date in the partial orders. Of course, this means that unlike in a release-based case, where write notices could be discarded by the releaser once they had been sent to all copies, in the acquire-based case a process must retain the write notices that it has either produced *or received* until it is certain that no later acquirer will need them. This can lead to significant storage overhead, and may require garbage collection techniques to keep the storage in check.

Multiple Writer with Release-based Coherence

With multiple writers, the type of coherence scheme we use depends on how the propagation of data (e.g. diffs) is managed; i.e. whether diffs are maintained at the multiple writers or propagated at a release to a fixed home. In either case, with release-based schemes a process needs only send write notices for the writes it has done since the previous release to all copies, and wait for acknowledgments for them at the next incoming acquire. However, since there is no single owner (writer) of a page at a given time, we cannot rely on an owner having an up to date copy of the sharing list. We need a mechanism like a directory to keep track of copies.

The next question is how does a process find the necessary data (diffs) when it incurs a page fault after an invalidation. If a home-based protocols is used to manage multiple writers, then this is easy: the page or diffs can be found at the home (the release must wait till the diffs reach the home before it completes). If diffs are maintained at the writers in a distributed form, the faulting process must know not only from where to obtain the diffs but also in what order they should be applied. This is because diffs for the same data may have been produced either in different intervals in the same process, or in intervals on different processes but in the same causal chain or acquires and releases; when they arrive at a processor, they must be applied in accordance with the partial orders needed for causality. The locations of the diffs may be determined from the incoming write notices. However, the order of application is difficult to determine without vector time-stamps. This is why when homes are not used for data, simple directory and release-based multiple writer coherence schemes use updates rather than invalidations as the write notices: The diffs themselves are sent to the sharers at the release. Since a release waits for acknowledgments before completing, there is no ordering problem in applying diffs: It is not possible for diffs for a page that correspond to different releases that are part of the same causal order to reach a processor at the same time. In other words, the diffs are guaranteed to reach processors exactly according to the desired partial order. This type of mechanism was used in the Munin system [CBZ91].

Version numbers are not useful with release-based multiple writer schemes. We cannot update version numbers at ownership changes, since there are none, but only at release points. And then version numbers don't help us save write notices: Since the releaser has just obtained the latest version number for the page at the release itself, there is no question of another node having a

more recent version number for that page. Also, since a releaser only has to send the write notices (or diffs) it has produced since its previous release, with an update-based protocol there is no need for vector time-stamps. If diffs were not sent out at a release but retained at the releaser, time-stamps would have been needed to ensure that diffs are applied in the correct order, as discussed above.

Multiple Writer with Acquire-based Coherence

The coherence issues and mechanisms here are very similar to the single-writer acquire-based schemes (the main difference is in how the data are managed). With no special support, the acquirer must obtain from the last releaser all write notices that it has received or produced since their last interaction, so large histories must be maintained and communicated. Version numbers can help reduce the number of invalidations applied, but not the number transferred (with home-based schemes, the version number can be incremented every time a diff gets to the home, but the releaser must wait to receive this number before it satisfies the next incoming acquire; without homes, a separate version manager must be designated anyway for a page, which causes complications). The best method is to use vector time-stamps as described earlier. The vector time-steps manage coherence for home-based schemes; for non home-based schemes, they also manage the obtaining and application of diffs in the right order (the order dictated by the time-stamps that came with the write notices).

To implement LRC, then, a processor has to maintain a lot of auxiliary data structures. These include:

*   For every page in its local memory an array, indexed by process, each entry being a list of write notices or diffs received from that process.

*   A separate array, indexed by process, each entry being a pointer to a list of interval records. These represent the intervals of that processor for which the current processor has already received write notices. An interval record points to the corresponding list of write notices, and each write notice points to its interval record.

*   A free pool for creating diffs.

Since these data structures may have to be kept around for a while, determined by the precedence order established at runtime, they can limit the sizes of problems that can be run and the scalability of the approach. Home-based approaches help reduce the storage for diffs: Diffs do not have to be retained at the releaser past the release, and the first array does not have to maintain lists of diffs received. Details of these mechanisms can be found in [KCD+94, ZIL97].

9.8 Concluding Remarks

The alternative approaches to supporting coherent replication in a shared address space discussed in this chapter raise many interesting sets of hardware-software tradeoffs. Relaxed memory models increase the burden on application software to label programs correctly, but allow compilers to perform their optimizations and hardware to exploit more low-level concurrency. COMA systems require greater hardware complexity but simplify the job of te programmer. Their effect on performance depends greatly on the characteristics of the application (i.e. whether sharing misses or capacity misses dominate inter-node communication). Finally, commodity-based approaches

that reduce system cost and provide a better incremental procurement model, but they often require substantially greater programming care to achieve good performance.

The approaches are also still controversial, and the tradeoffs have not shaken out. Relaxed memory models are very useful for compilers, but we will see in Chapter 11 that several modern processors are electing to implement sequential consistency at the hardware-software interface with increasingly sophisticated alternative techniques to obtain overlap and hide latency. Contracts between the programmer and the system have also not been very well integrated into current programming languages and compilers. Full hardware support for COMA was implemented in the KSR-1 [FBR93] but is not very popular in systems being built today because of to its high cost. However, approaches similar to Simple COMA are beginning to find acceptance in commercial products.

All-software approaches like SVM and fine-grained software access control have been demonstrated to achieve good performance at relatively small scale for some classes of applications. Because they are very easy to build and deploy, they will be likely be used on clusters of workstations and SMPs in several environments. However, the gap between these and all-hardware systems is still quite large in programmability as well as in performance on a wide range of applications, and their scalability has not yet been demonstrated. The commodity based approaches are still in the research stages, and it remains to be seen if they will become viable competitors to hardware cache-coherent machines for a large enough set of applications. It may also happen that the commoditization of hardware coherence assists and methods to integrate them into memory systems will make these all-software approaches more marginal, for use largely in those environments that do not wish to purchase parallel machines but rather use clusters of existing machines as shared address space multiprocessors when they are otherwise idle. In the short term, economic reasons might nonetheless provide all-software solutions with another likely role. Since vendors are likely to build tightly-coupled systems with only a few tens or a hundred nodes, connecting these hardware-coherent systems together with a software coherence layer may be the most viable way to construct very large machines that still support the coherent shared address space programming model. While supporting a coherent shared address space on scalable systems with physically distributed memory is well established as a desirable way to build systems, only time will tell how these alternative approaches will shake out.

9.9 References

[Adv93]   Sarita V. Adve. Designing Memory Consistency Models for Shared-Memory Multiprocessors, Ph.D. Thesis, University of Wisconsin-Madison, 1993. Available as Computer Sciences Technical Report #1198, University of Wisconsin-Madison, December 1993.

[AdH90a]   Sarita Adve and Mark Hill. Weak Ordering: A New Definition. In *Proceedings of 17th International Symposium on Computer Architecture*, pp. 2-14, May 1990.

[AdH90b]   Sarita Adve and Mark Hill. Implementing Sequential Consistency in Cache-Based Systems. In *Proceedings of 1990 International Conference on Parallel Processing*, pp. 47-50, August 1990. <<unused>>

[AdH93]   Sarita Adve and Mark Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613-624, June 1993. <<unused>>

[AGG+93]   Sarita Adve, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, and Mark Hill. Sufficient Systems Requirements for Supporting the PLpc Memory Model. Technical Report #1200, Com-

puter Sciences, University of Wisconsin - Madison, December 1993. Also available as Stanford University Technical Report CSL-TR-93-595.

[AnL93]   Jennifer Anderson and Monica Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993. <<unused>>

[BZS93]   Brian N. Bershad, Matthew J. Zekauskas and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In Proceedings of COMPCON'93, February 1993.

[BIS97]   Angelos Bilas, Liviu Iftode and Jaswinder Pal Singh. Evaluation of Hardware Support for Next-Generation Shared Virtual Memory Clusters. Technical Report no. CS-TR-97-xx, Computer Science Department, Princeton University.

[BR90]   R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of 17th International Symposium on Computer Architecture*, pp. 115-124, May 1990.

[BLA+94]   Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felten and Jonathan Sandberg. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *Proceedings of 21st International Symposium on Computer Architecture*, pp. 142-153, April 1994.

[BBG+93]   F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In Proceedings of Supercomputing'93, pp. 588-597, November 1993. See also *Scientific Programming*, vol. 2, no. 3, Fall 1993.

[CBZ91]   John B. Carter, J.K. Bennett and Willy Zwaenepoel. Implementation and Performance of Munin. In Proceedings of the Thirteenth Symposium on Operating Systems Principles, pp. 152-164, October, 1991.

[CBZ95]   John B. Carter, J. K. Bennett and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. In ACM Transactions of Computer Systems, vol. 13, no. 3, August 1995, pp. 205-244.

[CV90]   Hoichi Cheong and Alexander Viedenbaum. Compiler-directed Cache Management in Multiprocessors. IEEE Computer, vol. 23, no. 6, June 1990, p. 39-47.

[Con93]   Convex Computer Corporation. Exemplar Architecture. Convex Computer Corporation, Richardson, Texas. November 1993.

[CSB93]   Francisco Corella, Janice Stone, Charles Barton. A Formal Specification of the PowerPC Shared Memory Architecture. Technical Report Computer Science RC 18638 (81566), IBM Research Division, T.J. Watson Research Center, January 1993.

[CDG+93]   David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In Proceedings of Supercomputing'93, pp. 262-273, November 1993.

[DSB86]   Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of 13th International Symposium on Computer Architecture*, pp. 434-442, June 1986.

[DuS90]   Michel Dubois and Christoph Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660-673, June 1990.

[DWB+91]   Michel Dubois, J.-C. Wang, L.A. Barroso, K. Chen and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In Proceedings of Supercomputing'91, pp. 197-206, November 1991.

[DKC+93]   Sandhya Dwarkadas, Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of 20th International Symposium on Computer Architecture*, pp. 144-155, May 1993.

[ENC+96]   Andrew Erlichson, Neal Nuckolls, Greg Chesson and John Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 210--220, October, 1996.

[FLR+94]   Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers, and David A. Wood. Application-Specific Protocols for User-Level Shared Memory. In Proceedings of Supercomputing'94, November 1994.

[FaW97]   Babak Falsafi and David A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of 24th International Symposium on Computer Architecture*, June 1997.

[FBR93]   Steve Frank, Henry Burkhardt III, James Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of COMPCON*, pp. 285-294, Spring 1993.

[FCS+93]   Jean-Marc Frailong, et al. The Next Generation SPARC Multiprocessing System Architecture. In *Proceedings of COMPCON*, pp. 475-480, Spring 1993.

[GCP96]   R. Gillett and M. Collins and D. Pimm. Overview of Network Memory Channel for PCI. In Proceedings of the IEEE Spring COMPCON '96, February 1996.

[GLL+90]   Kourosh Gharachorloo, et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of 17th International Symposium on Computer Architecture*, pp. 15-26, May 1990.

[GGH91]   Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of 1991 International Conference on Parallel Processing*, pp. I:355-364, August 1991.

[GAG+92]   Kourosh Gharachorloo, et al. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399-407, August 1992.

[Gha95]   Kourosh Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Ph.D. Dissertation, Computer Systems Laboratory, Stanford University, December 1995.

[Goo89]   James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report No. 1006, Computer Science Department, University of Wisconsin-Madison, February 1989.

[Hag92]   Erik Hagersten. Toward Scalable Cache Only Memory Architectures. Ph.D. Dissertation, Swedish Institute of Computer Science, Sweden, October 1992.

[HLH92]   E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache Only Memory Architecture. *IEEE Computer*, vol. 25, no. 9, pp. 44-54, September 1992.

[HeP95]   John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach. Second Edition, Morgan Kaufmann Publishers, 1995.

[HKO+94]   Mark Heinrich, Jeff Kuskin, Dave Ofelt, John Heinlein, Joel Baxter, J.P. Singh, Rich Simoni, Kourosh Gharachorloo, Dave Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Performance Impact of Flexibility on the Stanford FLASH Multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 274-285, October 1994.

[HPF93   High Performance Fortran Forum. High Performance Fortran Language Specification. Scientific Programming, vol. 2, pp. 1-270, 1993.

[IDF+96]   L. Iftode and C. Dubnicki and E. W. Felten and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In Proceedings of the Second Symposium on High Performance Computer Architecture, February 1996.

[ISL96a]   Liviu Iftode, Jaswinder Pal Singh and Kai Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd International Symposium on Computer Ar-*

chitecture, April1996.

[ISL96b]   Liviu Iftode, Jaswinder Pal Singh and Kai Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In Proceedings of the Symposium on Parallel Algorithms and Architectures, June 1996.

[JSS97]   Dongming Jiang, Hongzhang Shan and Jaswinder Pal Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 217-229, June 1997.

[Joe95]   Truman Joe. COMA-F A Non-Hierarchical Cache Only Memory Architecture. Ph.D. thesis, Computer Systems Laboratory, Stanford University, March 1995.

[JoH94]   Truman Joe and John L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 82-93, April 1994.

[KCD+94]   Peter Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Proceedings of the Winter USENIX Conference, January 1994, pp. 15-132.

[KCZ92]   Peter Keleher and Alan L. Cox and Willy Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 13-21, May 1992.

[KHS+97]   Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michal Cierniak, Srinivasan Parthasarathy, Wagner Meira, Sandhya Dwarkadas and Michael Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[KoS96]   Leonidas. I. Kontothanassis and Michael. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory", In Proceedings of the Second Symposium on High Performance Computer Architecture, pp. ??, February 1996.

[KOH+94]   Jeff Kuskin, Dave Ofelt, Mark Heinrich, John Heinlein, Rich Simoni, Kourosh Gharachorloo, John Chapin, Dave Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 302-313, April 1994.

[LiH89]   Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In ACM Transactions on Computer Systems, vol. 7, no. 4, Nov. 1989, pp. 321--359.

[LS88]   Richard Lipton and Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technical report no. CS-TR-180-88, Computer Science Department, Princeton University, September 1988.

[LRV96]   James R. Larus, Brad Richards and Guhan Viswanathan. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In Gregory V. Wilson and Paul Lu, eds., Parallel Programming Using C++, MIT Press 1996.

[LLJ+93]   Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[MSS+94]   Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. The PowerPC Architecture: A Specification for a New Family of RISC Processors. Morgan Kaufman Publishers, Inc., 1994.

[RLW94]   Steven K. Reinhardt, James R. Larus and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of 21st International Symposium on Computer Architecture*, pp. 325-337, 1994.

[RPW96]   Steven K. Reinhardt, Robert W. Pfile and David A. Wood. Decoupled Hardware Support for Dis-

tributed Shared Memory. In *Proceedings of 23rd International Symposium on Computer Architecture*, 1996.

[RSL93]   Martin Rinard, Daniel Scales and Monica Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. IEEE Computer, June 1993.

[RSG93]   Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 14-25, May 1993. <<unused>>

[SBI+97]   Rudrojit Samanta, Angelos Bilas, Liviu Iftode and Jaswinder Pal Singh. Home-based SVM for SMP Clusters. Technical Report no. CS-TR-97-xx, Computer Science Department, Princeton University

[SWC+95]   Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument For Simple COMA. In Proceedings of the First IEEE Symposium on High Performance Computer Architecture, pp. 276-285, January 1995.

[SGT96]   Daniel J. Scales, Kourosh Gharachorloo and Chandramohan A. Thekkath. Shasta: A Low Overhead, SOftware-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[SFC91]   P. Sindhu, J.-M. Frailong, and M. Cekleov. Formal Specification of Memory Models. Technical Report (PARC) CSL-91-11, Xerox Corporation, Palo Alto Research Center, December 1991.

[SFL+94]   Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 297-307, 1994.

[ShS88]   Shasha, D. and Snir, M. Efficient and Correct Execution of Parallel Programs that Share Memory. ACM Transactions on Programming Languages and Operating Systems, vol. 10, no. 2, April 1988, pp. 282-312.

[Sit92]   Richard L. Sites, editor. Alpha Architecture Reference Manual. Digital Press, 1992.

[SJG92]   Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 80-91, May 1992.

[SDH+97]   Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srivinasan Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[Spa91]   The SPARC Architecture Manual. SUN Microsystems Inc., No. 800-199-12, Version 8, January 1991.

[TSS+96]   Radhika Thekkath, Amit Pal Singh, Jaswinder Pal Singh, John Hennessy and Susan John. An Application-Driven Evaluation of the Convex Exemplar SP-1200. *In Proceedings of the International Parallel Processing Symposium*, June 1997.

[WeG94]   David Weaver and Tom Germond, editors. The SPARC Architecture Manual. Prentice Hall, 1994. SPARC International, Version 9.

[WeS94]   Shlomo Weiss and James Smith. Power and PowerPC. Morgan Kaufmann Publishers Inc. 1994. <<unused>>

[ZIL96]   Yuanyuan Zhou, Liviu Iftode and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In Proceedings of the Operating Systems Design and Implementation Symposium, October 1996.

9.10 Exercises

9.1 General

a. Why are update-based coherence schemes relatively incompatible with the sequential consistency memory model in a directory-based machine?

b. The Intel Paragon machine discussed in Chapter 7 has two processing elements per node, one of which always executes in kernel mode to support communication. Could this processor be used effectively as a programmable communication assist to support cache coherence in hardware, like the programmable assist of the Stanford FLASH or the Sequent NUMA-Q?

9.2 Eager versus delayed replies to processor.

a. In the Origin protocol, with invalidation acknowledgments coming to the home, it would be possible for other read and write requests to come to the requestor that has outstanding invalidations before the acknowledgments for those invalidations come in.

(i) Is this possible or a problem with delayed exclusive replies? With eager exclusive replies? If it is a problem, what is the simplest way to solve it?

(ii) Suppose you did indeed want to allow the requestor with invalidations outstanding to process incoming read and write requests. Consider write requests first, and construct an example where this can lead to problems. (Hint: consider the case where P1 writes to a location A, P2 writes to location B which is on the same cache block as A and then writes a flag, and P3 spins on the flag and then reads the value of location B.) How would you allow the incoming write to be handled but still maintain correctness? Is it easier if invalidation acknowledgments are collected at the home or at the requestor?

(iii) Now answer the same questions above for incoming read requests.

(iv) What if you were using an update-based protocol. Now what complexities arise in allowing an incoming request to be processed for a block that has updates outstanding from a previous write, and how might you solve them?

(v) Overall, would you choose to allow incoming requests to be processed while invalidations or updates are outstanding, or deny them?

b. Suppose a block needs to be written back while there are invalidations pending for it. Can this lead to problems, or is it safe? If it is problematic, how might you address the problem?

c. Are eager exclusive replies useful with an underlying SC model? Are they at all useful if the processor itself provides out of order completion of memory operations, unlike the MIPS R10000?

d. Do the tradeoffs between collecting acknowledgments at the home and at the requestor change if eager exclusive replies are used instead of delayed exclusive replies?

9.3 Relaxed Memory Consistency Models

a. If the compiler reorders accesses according to WO, and the processor's memory model is SC, what is the consistency model at the programmer's interface? What if the compiler is SC and does not reorder memory operations, but the processor implements RMO?

b. In addition to reordering memory operations (reads and writes) as discussed in the example in the chapter, register allocation by a compiler can also eliminate memory accesses entirely. Consider the example code fragment shown below. Show how register allocation

can violate SC. Can a uniprocessor compiler do this? How would you prevent it in the compiler you normally use?

P1	P2
A = 1	while (flag == 0);
flag = 1	u = A.

   c. Consider all the system specifications discussed in the chapter. Order them in order of weakness, i.e. draw arcs between models such that an arc from model A to model B indicates that A is stronger than B, i.e. that is any execution that is correct under A is also correct under B, but not necessarily vice versa.

   d. Which of PC and TSO is better suited to update-based directory protocols and why?

   e. Can you describe a more relaxed system specification than Release Consistency, without explicitly associating data with synchronization as in Entry Consistency? Does it require additional programming care beyond RC?

   f. Can you describe a looser set of sufficient conditions for WO? For RC?

9.4 More Relaxed Consistency: Imposing orders through fences.

   a. Using the fence operations provided by the DEC Alpha and Sparc RMO consistency specifications, how would you ensure sufficient conditions for each of the RC, WO, PSO, TSO, and SC models? Which ones do you expect to be efficient and which ones not, and why?

   b. A *write-fence* operation stalls subsequent write operations until all of the processor's pending write operations have completed. A *full-fence* stalls the processor until **all** of its pending operations have completed.

   (i) Insert the minimum number of fence instructions into the following code to make it sequentially consistent. Don't use a full-fence when a write-fence will suffice.

```
ACQUIRE LOCK1
LOAD A
STORE A
RELEASE LOCK1
LOAD B
STORE B
ACQUIRE LOCK1
LOAD C
STORE C
RELEASE LOCK1
```

   (ii) Repeat part (i) to guarantee *release consistency.*

   c. The IBM-370 consistency model is much like TSO, except that it does not allow a read to return the value written by a previous write in program order until that write has completed. Given the following code segments, what combination of values for (u,v,w,x) are not allowed by SC. In each case, do the IBM-370, TSO, PSO, PC, RC and WO models preserve SC semantics without any ordering instructions or labels, or do they not? If not, insert the necessary fence operations to make them conform to SC. Assume that all variables had the value 0 before this code fragment was reached.

(i)

P1	P2
A = 1	B = 1
u = A	v = B
w = B	x = A

(ii)

P1	P2
A = 1	B = 1
C = 1	C = 2
u = C	v = C
w = B	x = A

    d. Consider a two-level coherence protocol with snooping-based SMPs connected by a memory-based directory protocol, uisng release consistency. While invalidation acknowledgments are still pending for a write to a memory block, is it okay to supply the data to another processor in (i) the same SMP node, or (ii) a different SMP node? Justify your answers and state any assumptions.

9.5 More Relaxed Consistency: Labeling and Performance

    a. Can a program that is not properly labeled run correctly on a system that supports release consistency? If so, how, and if not then why not?

    b. Why are there four flavor bits for memory barriers in the SUN Sparc V9 specification. Why not just two bits, one to wait for all previous writes to complete, and another for previous reads to complete?

    c. To communicate labeling information to the hardware (i.e. that a memory operations is labeled as an acquire or a release), there are two options: One is to associate the label with the address of the location, the other with the specific operation in the code. What are the tradeoffs between the two? [

    d. Two processors P1 and P2 are executing the following code fragments under the sequential consistency (SC) and release consistency (RC) models. Assume an architecture where both read and write misses take 100 cycles to complete. However, you can assume that accesses that are allowed to be overlapped under the consistency model are indeed overlapped. Grabbing a free lock or unlocking a lock takes 100 cycles, and no overlap is possible with lock/unlock operations from the same processor. Assume all the variables and locks are initially uncached and all locks are unlocked. All memory locations are initialized to 0. All memory locations are distinct and map to different indices in the caches.

```
P1                      P2

LOCK (L1)               LOCK(L1)

A = 1                   x = A

B = 2                   y = B

UNLOCK(L1)              x1 = A
```

```
                        UNLOCK(L1)

                        x2 = B
```

> (i) What are the possible outcomes for x and y under SC? What are the possible outcomes of x and y under RC?
>
> (ii) Assume P1 gets the lock first. After how much time will P2 complete under SC? Under RC?

e. Given the following code fragment, we want to compute its execution time under various memory consistency models. Assume a processor architecture with an arbitrarily deep write buffer. All instructions take 1 cycle ignoring memory system effects. Both read and write misses take 100 cycles to complete (i.e. globally perform). Locks are cacheable and loads are non-blocking. Assume all the variables and locks are initially uncached, and all locks are unlocked. Further assume that once a line is brought into the cache it does not get invalidated for the duration of the code's execution. All memory locations referenced

LOAD A
STORE B
LOCK (L1)
STORE C
LOAD D
UNLOCK (L1)
LOAD E
STORE F

> above are distinct; furthermore they all map to different cache lines.
>
> (i) If *sequential* consistency is maintained, how many cycles will it take to execute this code?
>
> (ii) Repeat part (i) for *weak* ordering.
>
> (iii) Repeat part (i) for *release* consistency.

f. The following code is executed on an aggressive dynamically scheduled processor. The processor can have multiple outstanding instructions, the hardware allows for multiple outstanding misses and the write buffer can hide store latencies (of course, the above features may only be used if allowed by the memory consistency model.)

```
Processor 1:
sendSpecial(int value) {
    A = 1;
    LOCK(L);
    C = D*3;
    E = F*10
    G = value;
    READY = 1;
    UNLOCK(L);
```

```
                                    }

                            Processor 2:
                            receiveSpecial() {
                               LOCK(L);
                               if (READY) {
                                   D = C+1;
                                   F = E*G;
                               }
                               UNLOCK(L);
                            }
```

Assume that locks are non-cachable and are acquired either 50 cycles from an issued request or 20 cycles from the time a release completes (whichever is later). A release takes 50 cycles to complete. Read hits take one cycle to complete and writes take one cycle to put into the write buffer. Read misses to shared variables take 50 cycles to complete. Writes take 50 cycles to complete. The write buffer on the processors is sufficiently large that it never fills completely. Only count the latencies of reads and writes to shared variables (those listed in capitals) and the locks. All shared variables are initially uncached with a value of 0. Assume that Processor 1 obtains the lock first.

(i) Under SC, how many cycles will it take from the time processor 1 enters sendSpecial() to the time that processor 2 leaves receiveSpecial()? Make sure that you justify your answer. Also make sure to note the issue and completion time of each synchronization event.

(ii) How many cycles will it take to return from receiveSpecial() under Release Consistency?

9.6 Shared Virtual Memory

a. In eager release consistency diffs are created and also propagated at release time, while in one form of lazy release consistency they are created at release time but propagated only at acquire time, i.e. when the acquire synchronization request comes to the processor.:

(i) Describe some other possibilities for when diffs might be created, propagated, and applied in all-software lazy release consistency (think of release time, acquire time, or access fault time). What is the laziest scheme you can design?

(ii) Construct examples where each lazier scheme leads to less diffs being created/communicated, and/or fewer page faults.

(iii) What complications does each lazier scheme cause in implementation, and which would you choose to implement?

b. Delaying the propagation of invalidations until a release point or even until the next acquire point (as in lazy release consistency) can be done in hardware-coherent systems as well. Why is LRC not used in hardware-coherent systems? Would delaying invalidations until a release be advantageous?

c. Suppose you had a co-processor to perform the creation and application of diffs in an all-software SVM system, and therefore did not have to perform this activity on the main processor. Considering eager release consistency and the lazy variants you designed above, comment on the extent to which protocol processing activity can be overlapped with computation on the main processor. Draw time-lines to show what can be done on the main processor and on the co-processor. Do you expect the savings in performance to be sub-

stantial? What do you think would be the major benefit and implementation complexity of having all protocol processing and management be performed on the co-processor?

d. Why is garbage collection more important and more complex in lazy release consistency than in eager release consistency? What about in home-based lazy release consistency? Design a scheme for periodic garbage collection (when and how), and discuss the complications.

9.7 Fine-grained Software Shared Memory

a. In systems like Blizzard-S or Shasta that instrument read and write operations in software to provide fine-grained access control, a key performance goal is to reduce the overhead of instrumentation. Describe some techniques that you might use to do this. To what extent do you think the techniques can be automated in a compiler or instrumentation tool?

b. When messages (e.g. page requests or lock requests) arrive at a node in a software shared memory system, whether fine-grained or coarse grained, there are two major ways to handle them in the absence of a programmable communication assist. One is to interrupt the main processor, and the other is to have the main processor poll for messages.

(i) What are the major tradeoffs between the two methods?

(ii) Which do you expect to perform better for page-based shared virtual memory and why? For fine-grained software shared memory? What application characteristics would most influence your decision?

(iii) What new issues arise and how might the tradeoffs change if each node is an SMP rather than a uniprocessor?

(iv) How do you think you would organize message handling with SMP nodes?

9.8 Shared Virtual Memory and Memory Consistency Models

a. List all the tradeoffs you can think of between LRC based on diffs (not home based) versus based on automatic update. Which do you think would perform better? What about home-based LRC based on diffs versus based on automatic update?

b. Is a properly labeled program guaranteed to run correctly under LRC? Under ERC? Under RC? Under Scope Consistency? Is a program that runs correctly under ERC guaranteed to run correctly under LRC? Is it guaranteed to be properly labeled (i.e. can a program that is not properly labeled run correctly under ERC? Is a program that runs correctly under LRC guaranteed to run correctly under ERC? Under Scope Consistency?

c. Consider a single-writer release-based protocol. On a release, does a node need to find the up-to-date sharing list for each page you've modified since the last release from the current owner, or just send write notices to nodes on your version of the sharing list for each such page? Explain why.

d. Consider version numbers without directories. Does this avoid the problem of sending multiple invalidates to the same copy. Explain why, or give a counter-example.

e. Construct an example where Scope Consistency transparently delivers performance improvement over regular home-based LRC, and one where it needs additional programming complexity for correctness; i.e. a program that satisfied release consistency but would not work correctly under scope consistency.

9.9 Alternative Approaches. Trace the path of a write reference in (i) a pure CC-NUMA, (ii) a flat COMA, (iii) an SVM with automatic update, (iv) an SVM protocol without automatic update, and (v) a simple COMA.

9.10 Software Implications and Performance Issues

a. You are performing an architectural study using four applications: Ocean, LU, an FFT that uses a matrix transposition between local calculations on rows, and Barnes-Hut. For each application, answer the above questions assuming a page-grained SVM system (these questions were asked for a CC-NUMA system in the previous chapter):

(i) what modifications or enhancements in data structuring or layout would you use to ensure good interactions with the extended memory hierarchy?

(ii) what are the interactions with cache size and granularities of allocation, coherence and communication that you would be particularly careful to represent or not represent? What new ones become important in SVM systems that were not so important in CC-NUMA?

(iii) are the interactions with cache size as important in SVM as they are in CC-NUMA? If they are of different importance, say why.

9.11 Consider the FFT calculation with a matrix transpose described in the exercises at the end of the previous chapter. Suppose you are running this program on a page-based SVM system using an all-software, home-based multiple-writer protocol.

a. Would you rather use the method in which a processor reads locally allocated data and writes remotely allocated data, or the one in which the processor reads remote data and writes local data?

b. Now suppose you have hardware support for automatic update propagation to further speed up your home-based protocol? How does this change the tradeoff, if at all?

c. What protocol optimizations can you think of that would substantially increase the performance of one scheme or the other?

CHAPTER 10   Interconnection Network Design

10.1   Introduction

We have seen throughout this book that scalable high-performance interconnection networks lie at the core of parallel computer architecture. Our generic parallel machine has basically three components: the processor-memory nodes, the node-to-network interface, and the network that holds it all together. The previous chapters give a general understanding of the requirements placed on the interconnection network of a parallel machine; this chapter examines the design of high-performance interconnection networks for parallel computers in depth. These networks share basic concepts and terminology with local area networks (LAN) and wide area networks (WAN), which may be familiar to many readers, but the design trade-offs are quite different because of the dramatic difference of time scale.

Parallel computer networks are a rich and interesting topic because they have so many facets, but this richness also make the topic difficult to understand in an overall sense. For example, parallel

computer networks are generally wired together in a regular pattern. The topological structure of these networks have elegant mathematical properties and there are deep relationships between these topologies and the fundamental communication patterns of important parallel algorithms. However, pseudo-random wiring patterns have a different set of nice mathematical properties and tend to have more uniform performance, without really good or really bad communication patterns. There is a wide range of interesting trade-offs to examine at this abstract level and a huge volume of research papers focus completely on this aspect of network design. On the other hand, passing information between two independent asynchronous devices across an electrical or optical link presents a host of subtle engineering issues. These are the kinds of issues that give rise to major standardization efforts. From yet a third point of view, the interactions between multiple flows of information competing for communications resources have subtle performance effects which are influenced by a host of factors. The performance modeling of networks is another huge area of theoretical and practical research. Real network designs address issues at each of these levels. The goal of this chapter is to provide a holistic understanding of the many facets of parallel computer networks, so the reader may see the large, diverse networks design space within the larger problem of parallel machine design as driven by application demands.

As with all other aspects of design, network design involves understanding trade-offs and making compromises so that the solution is near-optimal in a global sense, rather than optimized for a particular component of interest. The performance impact of the many interacting facets can be quite subtle. Moreover, there is not a clear consensus in the field on the appropriate cost model for networks, since trade-offs can be made between very different technologies, for example, bandwidth of the links may be traded against complexity of the switches. It is also very difficult to establish a well-defined workload against which to assess network designs, since program requirements are influenced by every other level of the system design before being presented to the network. This is the kind of situation that commonly gives rise to distinct design "camps" and rather heated debates, which often neglect to bring out the differences in base assumptions. In the course of developing the concepts and terms of computer networks, this chapter points out how the choice of cost model and workload lead to various important design points, which reflect key technological assumptions.

Previous chapters have illuminated the key driving factors on network design. The communication-to-computation ratio of the program places a requirement on the data *bandwidth* the network must deliver if the processors are to sustain a given computational rate. However, this load varies considerably between programs, the flow of information may be physically localized or dispersed, and it may be bursty in time or fairly uniform. In addition, the waiting time of the program is strongly affected by the *latency* of the network, and the time spent waiting affects the bandwidth requirement. We have seen that different programming models tend to communicate at different granularities, which impacts the size of data transfers seen by the network, and that they use different protocols at the network transaction level to realize the higher level programming model.

This chapter begins with a set of basic definitions and concepts that underlie all networks. Simple models of communication latency and bandwidth are developed to reveal the core differences in network design styles. Then the key components which are assembled to form networks are described concretely. Section 10.3 explains the rich space of interconnection topologies in a common framework. Section 10.4 ties the trade-offs back to cost, latency, and bandwidth under basic workload assumptions. Section 10.5 explains the various ways that messages are routed within the topology of the network in a manner that avoids deadlock and describes the further impact of routing on communication performance. Section 10.6 dives deeper into the hardware organiza-

tion of the switches that form the basic building block of networks in order to provide a more precise understanding of the inherent cost of various options and the mechanics underlying the more abstract network concepts. Section 10.7 explores the alternative approaches to flow control within a network. With this grounding in place, Section 10.8 brings together the entire range of issues in a collection of case studies and examines the transition of parallel computer network technology into other network regimes, including the emerging *System Area Network*s.

10.1.1 Basic definitions

The job of an interconnection network in a parallel machine is to transfer information from any source node to any desired destination node, in support of the network transactions that are used to realize the programming model. It should accomplish this task with as small a latency as possible, and it should allow a large number of such transfers to take place concurrently. In addition, it should be inexpensive relative to the cost of the rest of the machine.

The expanded diagram for our Generic Large-Scale Parallel Architecture in Figure 10-1 illustrates the structure of an interconnection network in a parallel machine. The communication assist on the source node initiates network transactions by pushing information through the network interface (NI). These transactions are handled by the communication assist, processor, or memory controller on the destination node, depending on the communication abstraction that is supported.
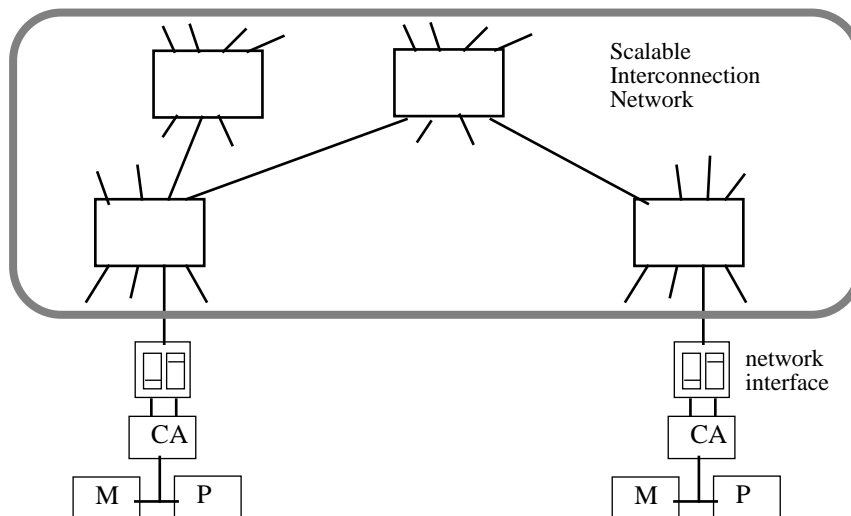
Figure 10-1 Generic Parallel Machine Interconnection Network

The communication assist initiates network transactions on behalf of the processor or memory controller, through a network interface that causes information to be transmitted across a sequence of links and switches to a remote node where the network transaction takes place.

The network is composed of links and switches, which provide a means to route the information from the source node to the destination node. A link is essentially a bundle of wires or a fiber that carries an analog signal. For information to flow along a link, a transmitter converts digital information at one end into an analog signal that is driven down the link and converted back into digi-

tal symbols by the receiver at the other end. The *physical* protocol for converting between streams of digital symbols and an analog signal forms the lowest layer of the network design. The transmitter, link, and receiver collectively form a *channel* for digital information flow between switches (or NIs) attached to the link. The *link level* protocol segments the stream of symbols crossing a channel into larger logical units, called packets or messages, that are interpreted by the switches in order to steer each packet or message arriving on an input channel to the appropriate output channel. Processing nodes communicate across a sequence of links and switches. The *node level protocol* embeds commands for the remote communication assist within the packets or messages exchanged between the nodes to accomplish network transactions.

Formally, a parallel machine interconnection network is a graph, where the vertices, V, are processing hosts or switch elements connected by communication *channels*, $C \subseteq V \times V$. A channel is a physical link between host or switch elements and a *buffer* to hold data as it is being transferred. It has a width, w, and a signalling rate, $f = \frac{1}{\tau}$, which together determine the *channel bandwidth*, $b = w \cdot f$. The amount of data transferred across a link in a cycle[1] is called a physical unit, or *phit*. Switches connect a fixed number of input channels to a fixed number of output channels; this number is called the switch *degree*. Hosts typically connect to a single switch, but can be multiply connected with separate channels. Messages are transferred through the network from a source host node to a destination host along a path, or *route*, comprised of a sequence of channels and switches.

A useful analogy to keep in mind is a roadway system composed of streets and intersections. Each street has a speed limit and a number of lanes, determining its peak bandwidth. It may be either unidirectional (one-way) or bidirectional. Intersections allow travelers to switch among a fixed number of streets. In each trip a collection of people travel from a source location along a route to a destination. There are many potential routes they may use, many modes of transportation, and many different ways of dealing with traffic encountered en route. A very large number of such "trips" may be in progress in a city concurrently, and their respective paths may cross or share segments of the route.

A network is characterized by its *topology, routing algorithm, switching strategy*, and *flow control* mechanism.

- The **topology** is the physical interconnection structure of the network graph; this may be regular, as with a two-dimensional grid (typical of many metropolitan centers), or it may be irregular. Most parallel machines employ highly regular networks. A distinction is often made between *direct* and *indirect* networks; direct networks have a host node connected to each switch, whereas indirect networks have hosts connected only to a specific subset of the switches, which form the edges of the network. Many machines employ hybrid strategies, so the more critical distinction is between the two types of nodes: hosts generate and remove

---

1. Since many networks operate asynchronously, rather than being controlled by a single global clock, the notion of a network "cycle" is not as widely used as in dealing with processors. One could equivalently define the network cycle time as the time to transmit the smallest physical unit of information, a phit.

For parallel architectures, it is convenient to think about the processor cycle time and the network cycle time in common terms. Indeed, the two technological regimes are becoming more similar with time.

traffic, whereas switches only move traffic along. An important property of a topology is the *diameter* of a network, which is the maximum shortest path between any two nodes.

- The **routing algorithm** determines *which* routes messages may follow through the network graph. The routing algorithm restricts the set of possible paths to a smaller set of legal paths. There are many different routing algorithms, providing different guarantees and offering different performance trade-offs. For example, continuing the traffic analogy, a city might eliminate gridlock by legislating that cars must travel east-west before making a single turn north or south toward their destination, rather than being allowed to zig-zag across town. We will see that this does, indeed, eliminate deadlock, but it limits a driver's ability to avoid traffic en route. In a parallel machine, we are only concerned with routes from a host to a host.

- The **switching strategy** determines *how* the data in a message traverses its route. There are basically two switching strategies. In *circuit switching* the path from the source to the destination is established and reserved until the message is transferred over the circuit. (This strategy is like reserving a parade route; it is good for moving a lot of people through, but advanced planning is required and it tends to be unpleasant for any traffic that might cross or share a portion of the reserved route, even when the parade is not in sight. It is also the strategy used in phone systems, which establish a circuit through possibly many switches for each call.) The alternative is *packet switching*, in which the message is broken into a sequence of *packets*. A packet contains routing and sequencing information, as well as data. Packets are individually routed from the source to the destination. (The analogy to traveling as small groups in individual cars is obvious.) Packet switching typically allows better utilization of network resources because links and buffers are only occupied while a packet is traversing them.

- The **flow control mechanism** determines *when* the message, or portions of it, move along its route. In particular, flow control is necessary whenever two or more messages attempt to use the same network resource, *e.g.*, a channel, at the same time. One of the traffic flows could be stalled in place, shunted into buffers, detoured to an alternate route, or simply discarded. Each of these options place specific requirements on the design of the switch and also influence other aspects of the communication subsystem. (Discarding traffic is clearly unacceptable in our traffic analogy.) The minimum unit of information that can be transferred across a link and either accepted or rejected is called a flow control unit, or *flit*. It may be as small as a phit or as large as a packet or message.

To illustrate the difference between a phit and a flit, consider the nCUBE case study in Section 7.2.4. The links are a single bit wide so a phit is one bit. However, a switch accepts incoming messages in chunks of 36 bits (32 bits of data plus four parity bits). It only allows the next 36 bits to come in when it has a buffer to hold it, so the flit is 36 bits. In other machines, such as the T3D, the phit and flit are the same.

The *diameter* of a network is the maximum distance between two nodes in the graph. The *routing distance* between a pair of nodes is the number of links traversed en route; this is at least as large as the shortest path between the nodes and may be larger. The *average distance* is simply the average of the routing distance over all pairs of node; this is also the expected distance between a random pair of nodes. In a direct network routes must be provided between every pair of switches, whereas in an indirect network it is only required that routes are provided between hosts. A network is *partitioned* if a set of links or switches are removed such that some nodes are no longer connected by routes.

Most of the discussion in this chapter centers on packets, because packet switching is used in most modern parallel machine networks. Where specific important properties of circuit switching

arise, they are pointed out. A packet is a self-delimiting sequence of digital symbols and logically consists of three parts, illustrated in Figure 10-2: a *header*, a data *payload*, and a *trailer*. Usually the routing and control information comes in the header at the front of the packet so that the switches and network interface can determine what to do with the packet as it arrives. Typically, the error code is in the trailer, so it can be generated as the message spools out, onto the link. The header may also have a separate error checking code.

The two basic mechanisms for building abstractions in the context of networks are *encapsulation* and *fragmentation*. Encapsulation involves carrying higher level protocol information in an uninterpreted form within the message format of a given level. Fragmentation involves splitting the higher level protocol information into a sequence of messages at a given level. While these basic mechanisms are present in any network, in parallel computer networks the layers of abstraction tend to be much shallower than in, say, the internet, and designed to fit together very efficiently. To make these notions concrete, observe that the header and trailer of a packet form an "envelope" that encapsulates the data payload. Information associated with the node-level protocol is contained within this payload. For example, a read request is typically conveyed to a remote memory controller in a single packet and the cache line response is a single packet. The memory controllers are not concerned with the actual route followed by the packet, or the format of the header and trailer. At the same time, the network is not concerned with the format of the remote read request within the packet payload. A large bulk data transferred would typically not be carried out as a single packet, instead, it would be fragmented into several packets. Each would need to contain information to indicate where its data should be deposited or which fragments in the sequence it represents. In addition, a single packet is fragmented at the link level into a series of symbols, which are transmitted in order across the link, so there is no need for sequencing information. This situation in which higher level information is carried within an envelope that is interpreted by the lower level protocol, or multiple such envelopes, occurs at every level of network design.

Sequence of symbols transmitted over a channel
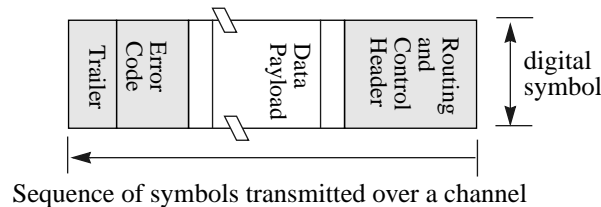
Figure 10-2 Typical Packet Format

A packet forms the logical unit of information that is steered along a route by switches. It is comprised of three parts, a header, data payload, and trailer. The header and trailer are interpreted by the switches as the packet progresses along the route, but the payload is not. The node level protocol is carried in the payload.

10.1.2 Basic communication performance

There is much to understand on each of the four major aspects of network design, but before going to these aspects in detail it is useful to have a general understanding of how they interact to determine the performance and functionality of the overall communication subsystem. Building on the brief discussion of networks in Chapter 5, let us look at performance from the latency and bandwidth perspectives.

Latency

To establish a basic performance model for understanding networks, we may expand the model for the communication time from Chapter 1. The time to transfer a single n-bytes of information from its source to its destination has four components, as follows.

$$Time(n)_{S-D} = Overhead + Routing\ Delay + Channel\ Occupancy + Contention\ Delay \qquad \textbf{(EQ 10.1)}$$

The *overhead* associated with getting the message into and out of the network on the ends of the actual transmission has been discussed extensively in dealing with the node-to-network interface in previous chapters. We have seen machines that are designed to move cache line sized chunks and others that are optimized for large message DMA transfers. In previous chapters, the routing delay and channel occupancy are effectively lumped together as the *unloaded latency* of the network for typical message sizes, and contention has been largely ignored. These other components are the focus of this chapter.

The channel occupancy provides a convenient lower bound on the communication latency, independent of where the message is going or what else is happening in the network. As we look at network design in more depth, we see below that the occupancy of each link is influenced by the channel width, the signalling rate, and the amount of control information, which is in turn influenced by the topology and routing algorithm. Whereas previous chapters were concerned with the *channel occupancy* seen from "outside" the network, the time to transfer the message across the bottleneck channel in the route, the view from within the network is that there is a channel occupancy associated with each step along the route. The communication assist is occupied for a period of time accepting the communication request from the processor or memory controller and spooling a packet into the network. Each channel the packet crosses en route is occupied for a period of time by the packet, as is the destination communication assist.

For example, an issue that we need to be aware of is the efficiency of the packet encoding. The packet envelop increases the occupancy because header and trailer symbols are added by the source and stripped off by the destination. Thus, the occupancy of a channel is $\dfrac{n + n_E}{b}$, where n_E is the size of the envelope and b is the raw bandwidth of the channel. This issue is addressed in the "outside view" by specifying the *effective bandwidth* of the link, derated from the raw bandwidth by $\dfrac{n}{n + n_E}$, at least for fixed sized packets. However, within the network packet efficiency remains a design issue. The effect is more pronounced with small packets, but it also depends on how routing is performed.

The *routing delay* is seen from outside the network as the time to move a given symbol, say the first byte of the message, from the source to the destination. Viewed from within the network, each step along the route incurs a routing delay that accumulates into the delay observed from the outside. The routing delay is a function of the number of channels on the route, called the *routing distance*, h, and the delay, Δ incurred as each switch as part of selecting the correct output port. (It is convenient to view the node-to-network interface as contributing to the routing delay like a switch.) The routing distance depends on the network topology, the routing algorithm, and the particular pair of source and destination nodes. The overall delay is strongly affected by switching and routing strategies.

With packet switched, *store-and-forward* routing, the entire packet is received by a switch before it is forwarded on the next link, as illustrated in Figure 10-3a. This strategy is used in most wide-area networks and was used in several early parallel computers. The unloaded network latency for an n byte packet, including envelope, with store-and-forward routing is

$$T_{sf}(n, h) = h\left(\frac{n}{b} + \Delta\right).$$

(EQ 10.2)

Equation 10.2 would suggest that the network topology is paramount in determining network latency, since the topology fundamentally determines the routing distance, h. In fact, the story is more complicated.

First, consider the switching strategy. With circuit switching one expects a delay proportional to h to establish the circuit, configure each of the switches along the route, and inform the source that the route is established. After this time, the data should move along the circuit in time n/b plus an additional small delay proportional to h. Thus, the unloaded latency in units of the network cycle time, τ, for an n byte message traveling distance h in a circuit switched network is:

$$T_{cs}(n, h) = \frac{n}{b} + h\Delta \ .$$

(EQ 10.3)

In Equation 10.3, the routing delay is an additive term, independent of the size of the message. Thus, as the message length increases, the routing distance, and hence the topology, becomes an insignificant fraction of the unloaded communication latency. Circuit switching is traditionally used in telecommunications networks, since the call set-up is short compared to the duration of the call. It is used in a minority of parallel computer networks, including the Meiko CS-2. One important difference is that in parallel machines the circuit is established by routing the message through the network and holding open the route as a circuit. The more traditional approach is to compute the route on the side and configure the switches, then transmit information on the circuit.

It is also possible to retain packet switching and yet reduce the unloaded latency from that of naive store-and-forward routing. The key concern with Equation 10.2 is that the delay is the product of the routing distance and the occupancy for the full message. However, a long message can be fragmented into several small packets, which flow through the network in a pipelined fashion. In this case, the unloaded latency is

$$T_{sf}(n, h, n_p) = \frac{n - n_p}{b} + h\left(\frac{n_p}{b} + \Delta\right),$$

(EQ 10.4)

where n_p is the size of the fragments. The effective routing delay is proportional to the packet size, rather than the message size. This is basically the approach adopted (in software) for traditional data communication networks, such as the internet.

In parallel computer networks, the idea of pipelining the routing and communication is carried much further. Most parallel machines use packet switching with *cut-through* routing, in which the switch makes its routing decision after inspecting only the first few phits of the header and allows the remainder of the packet to "cut through" from the input channel to the output channel, as indicated by Figure 10-3b. (If we may return to our vehicle analogy, cut-through routing is like what happens when a train encounters a switch in the track. The first car is directed to the proper

output track and the remainder follow along behind. By contrast, store-and-forward routing is like what happens at the station, where all the cars in the train must have arrived before the first can proceed toward the next station.) For cut-through routing, the unloaded latency has the same form as the circuit switch case,

$$T_{ct}(n, h) = \frac{n}{b} + h\Delta .$$ **(EQ 10.5)**

although the routing coefficients may differ since the mechanics of the process are rather different. Observe that with cut-through routing a single message may occupy the entire route from the source to the destination, much like circuit switching. The head of the message establishes the route as it moves toward its destination and the route clears as the tail moves through.

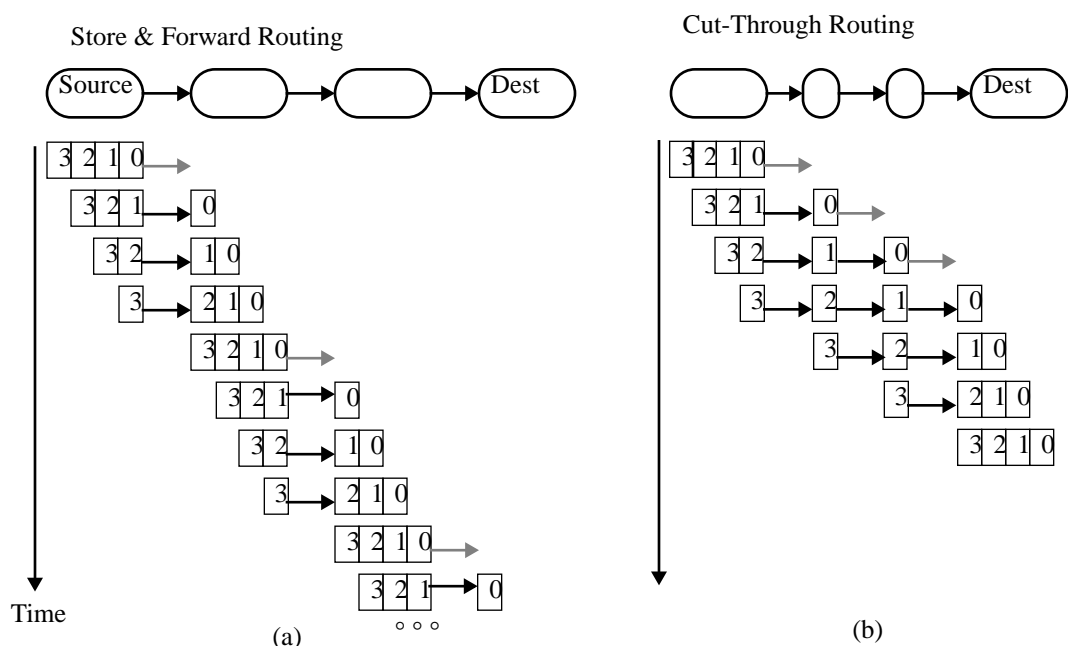Store & Forward Routing

Cut-Through Routing

Time

(a)

(b)

Figure 10-3 Store-and-forward vs. Cut-through routing

A four flit packet traverses three hops from source to destination under store&forward and cut-through routing. Cut-through achieves lower latency by making the routing decision (grey arrow) on the first flit and pipelining the packet through a sequence of switches. Store&forward accumulates the entire packer before routing it toward the destination.

The discussion of communication latency above addresses a message flowing from source to destination without running into traffic along the way. In this unloaded case, the network can be viewed simply as a pipeline, with a start-up cost, pipeline depth, and time per stage. The different switching and routing strategies change the effective structure of the pipeline, whereas the topology, link bandwidth, and fragmentation determine the depth and time per stage. Of course, the reason that networks are so interesting is that they are not a simple pipeline, but rather an interwoven fabric of portions of many pipelines. The whole motivation for using a network, rather than a bus, is to allow multiple data transfers to occur simultaneously. This means that one message flow may collide with others and contend for resources. Fundamentally, the network must provide a mechanism for dealing with contention. The behavior under contention depends on

several facets of the network design: the topology, the switching strategy, and the routing algorithm, but the bottom line is that at any given time a channel can only be occupied by one message. If two messages attempt to use the same channel at once, one must be deferred. Typically, each switch provides some means of arbitration for the output channels. Thus, a switch will select one of the incoming packets contending for each output and the others will be deferred in some manner.

An overarching design issue for networks is how contention is handled. This issue recurs throughout the chapter, so let's first just consider what it means for latency. Clearly, contention increases the communication latency experienced by the nodes. Exactly how it increases depends on the mechanisms used for dealing with contention within the network, which in turn differ depending on the basic network design strategy. For example, with packet switching, contention may be experienced at each switch. Using store-and-forward routing, if multiple packets that are buffered in the switch need to use the same output channel, one will be selected and the others are *blocked* in buffers until they are selected. Thus, contention adds queueing delays to the basic routing delay. With circuit switching, the effect of contention arises when trying to establish a circuit; typically, a routing probe is extended toward the destination and if it encounters a reserved channel, it is retracted. The network interface retries establishing the circuit after some delay. Thus, the start-up cost in gaining access to the network increases under contention, but once the circuit is established the transmission proceeds at full speed for the entire message.

With cut-through packet switching, two packet blocking options are available. The *virtual cut-through* approach is to spool the blocked incoming packet into a buffer, so the behavior under contention degrades to that of store-and-forward routing. The *worm-hole* approach buffers only a few phits in the switch and leaves the tail of the message in-place along the route. The blocked portion of the message is very much like a circuit held open through a portion of the network.

Switches have limited buffering for packets, so under sustained contention the buffers in a switch may fill up, regardless of routing strategy. What happens to incoming packets if there is no buffer space to hold them in the switch? In traditional data communication networks the links are long and there is little feedback between the two ends of the channel, so the typical approach is to *discard* the packet. Thus, under contention the network becomes highly unreliable and sophisticated protocols (e.g., TCP/IP) are used at the nodes to adapt the requested communication load to what the network can deliver without a high loss rate. Discarding on buffer overrun it also used with most ATM switches, even if they are employed to build closely integrated clusters. Like the wide area case, the source receives no indication that its packet was dropped, so it must rely on some kind of timeout mechanism to deduce that a problem has occured.

In most parallel computer networks, a packet headed for a full buffer is typically blocked in place, rather than discarded; this requires a handshake between the output port and input port across the link, i.e, link level flow control. Under sustained congestion, traffic 'backs-up' from the point in the network where contention for resources occurs towards the sources that are driving traffic into that point. Eventually, the sources experience back-pressure from the network (when it refuses to accept packets), which causes the flow of data into the network to slow down to that which can move through the bottleneck.[1] Increasing the amount of buffering within the network allows contention to persist longer without causing back-pressure at the source, but it also increases the potential queuing delays within the network when contention does occur.

One of the concerns that arises in networks with message blocking is that the 'back up' can impact traffic that is not headed for the highly contended output. Suppose that the network traffic

favors one of the destinations, say because it holds an important global variable that is widely used. This is often called a "hot spot." If the total amount of traffic destined for that output exceeds the bandwidth of the output, this traffic will back up within the network. If this condition persists, the backlog will propagate backward through the tree of channels directed at this destination, which is called *tree saturation*. [PfNo85]. Any traffic that cross the tree will also be delayed. In a worm-hole routed network, an interesting alternative to blocking the message is to discard it, but to inform the source of the collision. For example, in the BBN Butterfly the source held onto the tail of the message until the head reached the destination and if a collision occured in route, the worm was retracted all the way back to the source[ReTh86].

We can see from this brief discussion that all aspects of network design – link bandwidth, topology, switching strategy, routing algorithm, and flow control – combine to determine the latency per message. It should also be clear that there is a relationship between latency and bandwidth. If the communication bandwidth demanded by the program is low compared to the available network bandwidth, there will be few collisions, buffers will tend to be empty, and the latency will stay low, especially with cut-through routing. As the bandwidth demand increases, latency will increase due to contention.

An important point that is often overlooked is that parallel computer networks are effectively a *closed system*. The load placed on the network depends on the rate at which processing nodes request communication, which in turn depends on how fast the network delivers this communication. Program performance is affected most strongly by latency some kind of dependence is involved; the program must wait until a read completes or until a message is received to continue. While it waits, the load placed on the network drops and the latency decreases. This situation is very different from that of file transfers across the country contending with unrelated traffic. In a parallel machine, a program largely contends with itself for communication resources. If the machine is used in a multiprogrammed fashion, parallel programs may also contend with one another, but the request rate of each will be reduced as the serviced rate of the network is reduced due to the contention. Since low latency communication is critical to parallel program performance, the emphasis in this chapter is on cut-through packet-switched networks.

Bandwidth

Network bandwidth is critical to parallel program performance in part because higher bandwidth decreases occupancy, in part because higher bandwidth reduces the likelihood of contention, and in part because phases of a program may push a large volume of data around without waiting for individual data items to be transmitted along the way. Since networks behave something like pipelines it is possible to deliver high bandwidth even when the latency is large.

---

1. This situation is exactly like multiple lanes of traffic converging on a narrow tunnel or bridge. When the traffic flow is less than the flow-rate of the tunnel, there is almost no delay, but when the in-bound flow exceeds this bandwidth traffic backs-up. When the traffic jam fills the available storage capacity of the roadway, the aggregate traffic moves forward slowly enough that the aggregate bandwidth is equal to that of the tunnel. Indeed, upon reaching the mouth of the tunnel traffic accelerates, as the Bernoulli effect would require.

It is useful to look at bandwidth from two points of view: the "global" aggregate bandwidth available to all the nodes through the network and the "local" individual bandwidth available to a node. If the total communication volume of a program is M bytes and the aggregate communication bandwidth of the network is B bytes per second, then clearly the communication time is at least $\frac{M}{B}$ seconds. On the other hand, if all of the communication is to or from a single node , this estimate is far too optimistic; the communication time would be determined by the bandwidth through that single node.

Let us look first at the bandwidth available to a single node and see how it may be influenced by network design choices. We have seen that the effective local bandwidth is reduced from the raw link bandwidth by the density of the packet, $b\dfrac{n}{n+n_E}$. Furthermore, if the switch blocks the packet for the routing delay of Δ cycles while it makes its routing decision, then the effective local bandwidth is further derated to $b\left(\dfrac{n}{n+n_E+\Delta w}\right)$, since Δw is the opportunity to transmit data that is lost while the link is blocked. Thus, network design issues such as the packet format and the routing algorithm will influence the bandwidth seen by even a single node. If multiple nodes are communicating at once and contention arises, the perceived local bandwidth will drop further (and the latency will rise). This happens in any network if multiple nodes send messages to the same node, but it may occur within the interior of the network as well. The choice of network topology and routing algorithm affect the likelihood of contention within the network.

If many of the nodes are communicating at once, it is useful to focus on the global bandwidth that the network can support, rather than the bandwidth available to each individual node. First, we should sharpen the concept of the aggregate communication bandwidth of a network. The most common notion of aggregate bandwidth is the *bisection bandwidth*: of the network, which is the sum of the bandwidths of the minimum set of channels which, if removed, partition the network into two equal sets of nodes. This is a valuable concept, because if the communication pattern is completely uniform, half of the messages are expected to cross the bisection in each direction. We will see below that the bisection bandwidth per node varies dramatically in different network topologies. However, bisection bandwidth is not entirely satisfactory as a metric of aggregate network bandwidth, because communication is not necessarily distributed uniformly over the entire machine. If communication is localized, rather than uniform, bisection bandwidth will give a pessimistic estimate of communication time. An alternative notion of global bandwidth that caters to localized communication patterns would be the sum of the bandwidth of the links from the nodes to the network. The concern with this notion of global bandwidth, is that the internal structure of the network may not support it. Clearly, the available aggregate bandwidth of the network depends on the communication pattern; in particular, it depends on how far the packets travel, so we should look at this relationship more closely.

The total bandwidth of all the channels (or links) in the network is the number of channels times the bandwidth per channel, i.e., Cb bytes per second, Cw bits per cycle, or C phits per cycle. If each of N hosts issues a packet every M cycles with an average routing distance of h , a packet occupies, on average, h channels for $l = \frac{n}{w}$ cycles. Thus, he total load on the network is $\frac{Nhl}{M}$ phits per cycle. The average link utilization is at least

$$\rho = M\frac{C}{Nhl}, \qquad\qquad \text{(EQ 10.6)}$$

which obviously must be less than one. One way of looking at this is that the number of links per node, $\frac{C}{N}$, reflects the communication bandwidth (flits per cycle per node) available, on average, to each node. This bandwidth is consumed in direct proportion to the routing distance and the message size. The number of links per node is a static property of the topology. The average routing distance is determined by the topology, the routing algorithm, the program communication pattern, and the mapping of the program onto the machine. Good communication locality may yield a small h, while random communication will travel the average distance, and really bad patterns may traverse the full diameter. The message size is determined by the program behavior and the communication abstraction. In general, the aggregate communication requirements in Equation 10.6 says that as the machine is scaled up, the network resources per node must scale with the increase in expected latency.

In practice, several factors limit the channel utilization, ρ, well below unity. The load may not be perfectly balanced over all the links. Even if it is balanced, the routing algorithm may prevent all the links from being used for the particular communication pattern employed in the program. Even if all the links are usable and the load is balanced over the duration, stochastic variations in the load and contention for low-level resources may arise. The result of these factors is networks have a *saturation point*, representing the total channel bandwidth they can usefully deliver. As illustrated in Figure 10-4, if the bandwidth demand placed on the network by the processors is moderate, the latency remains low and the delivered bandwidth increases with increased demanded. However, at some point, demanding more bandwidth only increases the contention for resources and the latency increases dramatically. The network is essentially moving as much traffic as it can, so additional requests just get queued up in the buffers. One attempts to design parallel machines so that the network stays out of saturation, either by providing ample communication bandwidth or by limiting the demands placed by the processors.

A word of caution is in order regarding the dramatic increase in latency illustrated in Figure 10-4 as the network load approaches saturation. The behavior illustrated in the figure is typical of all queueing systems (and networks) under the assumption that the load placed on the system is independent of the response time. The sources keep pushing messages into the system faster than it can service them, so a queue of arbitrary length builds up somewhere and the latency grows with the length of this queue. In other words, this simple analysis assumes an *open system*. In reality, parallel machines are *closed systems*. There is only a limited amount of buffering in the network and, usually, only a limited amount of communication buffering in the network interfaces. Thus, if these "queues" fill up, the sources will slow down to the service rate, since there is no place to put the next packet until one is removed. The flow-control mechanisms affect this coupling between source and sink. Moreover, dependences within the parallel programs inherently embed some degree of end-to-end flow-control, because a processor must receive remote information before it can do additional work which depends on the information and generate additional communication traffic. Nonetheless, it is important to recognize that a shared resource, such as a network link, is not expected to be 100% utilized.

This brief performance modeling of parallel machine networks shows that the latency and bandwidth of real networks depends on all aspects of the network design, which we will examine in some detail in the remainder of the chapter. Performance modeling of networks itself is a rich
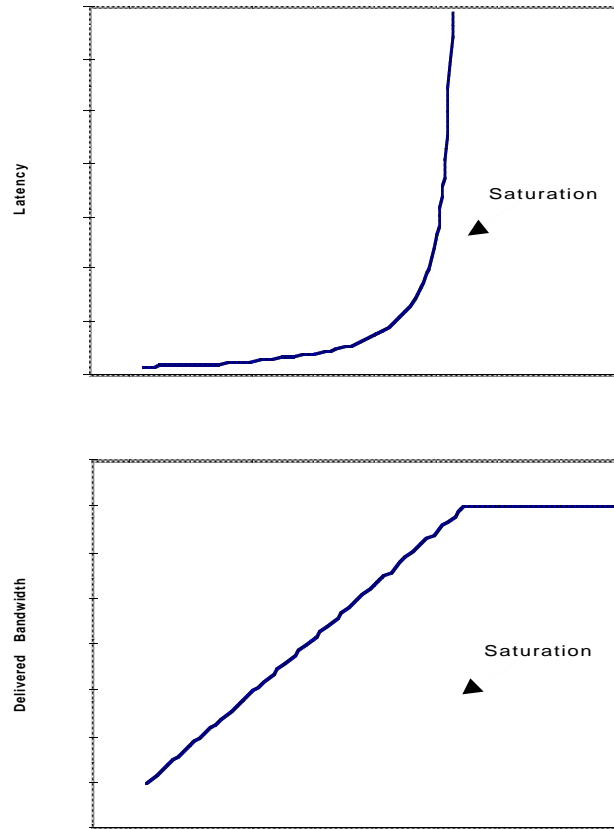
Figure 10-4 Typical Network Saturation Behavior

Networks can provide low latency when the requested bandwidth is well below that which can be delivered. In this regime, the delivered bandwidth scales linearly with that requested. However, at some point the network saturates and additional load causes the latency to increase sharply without yielding additional delivered bandwidth.

area with a voluminous literature base, and the interested reader should consult the references at the end of this chapter as a starting point. In addition, it is important to note that performance is not the only driving factor in network designs. Cost and fault tolerance are two other critical criteria. For example, the wiring complexity of the network is a critical issue in several large scale machines. As these issues depend quite strongly on specifics of the design and the technology employed, we will discuss them along with examining the design alternatives.

10.2 Organizational Structure

This section outlines the basic organizational structure of a parallel computer network. It is useful to think of this issue in the more familiar terms of the organization of the processor and the applicable engineering constraints. We normally think of the processor as being composed of datapath, control, and memory interface, including perhaps the on-chip portions of the cache hierarchy. The datapath is further broken down into ALU, register file, pipeline latches, and so

forth. The control logic is built up from examining the data transfers that take place in the datapath. Local connections within the datapath are short and scale well with improvements in VLSI technology, whereas control wires and busses are long and become slower relative to gates as chip density increases. A very similar notion of decomposition and assembly applies to the network. Scalable interconnection networks are composed of three basic components: links, switches, and network interfaces. A basic understanding of these components, their performance characteristics, and inherent costs is essential for evaluating network design alternatives. The set of operations is quite limited, since they fundamentally move packets toward their intended destination.

10.2.1 Links

A link is a cable of one or more electrical wires or optical fibers with a connector at each end attached to a switch or network interface port. It allows an analog signal to be transmitted from one end, received at the other, and sampled to obtain the original digital information stream. In practice, there is tremendous variation in the electrical and physical engineering of links, however, their essential logical properties can be characterized along three independent dimensions: length, width, clocking.

- A "short" link is one in which only a single logical value can be on the link at any time; a "long" link is viewed as a transmission line, where a series of logical values propagate along the link at a fraction of the speed of light, (1-2 feet per ns, depending on the specific medium).

- A "thin" link is one in which data, control, and timing information are multiplexed onto each wire, such as on a single serial link. A "wide" link is one which can simultaneously transmit data and control information. In either case, network links are typically narrow compared to internal processor datapath, say 4 to 16 data bits.

- The clocking may be "synchronous" or "asynchronous." In the synchronous case, the source and destination operate on the same global clock, so data is sampled at the receiving end according to the common clock. In the asynchronous case, the source encodes its clock in some manner within the analog signal that is transmitted and the destination recovers the source clock from the signal and transfers the information into its own clock domain.

A short electrical link behaves like a conventional connection between digital components. The signalling rate is determined by the time to charge the capacitance of the wire until it represents a logical value on both ends. This time increases only logarithmicaly with length if enough power can be used to drive the link.[1] Also, the wire must be terminated properly to avoid reflections, which is why it is important that the link be point-to-point.

The Cray T3D is a good example of a wide, short, synchronous link design. Each bidirectional link contains 24 bits in each direction, 16 for data, 4 for control, and 4 providing flow control for the link in the reverse direction, so a switch will not try to deliver flits into a full buffer. The entire machine operates under a single 150 MHz clock. A flit is a single phit of 16 bits. Two of the con-

---

1. The RC delay of a wire increases with the square of the length, so for a fixed amount of signal drive the network cycle time is strongly affected by length. However, if the driver strength is increased using an ideal driver tree, the time to drive the load of a longer wire only increases logarithmically. (If τ_{inv} is the propagation delay of a basic gate, then the effective propagation delay of a "short" wire of length l grows as $t_s = K \tau_{inv} \log l$.)

trol bits identify the phit type (00 No info, 01 Routing tag, 10 packet, 11 last). The routing tag phit and the last-packet phit provide packet framing.

In a long wire or optical fiber the signal propagates along the link from source to destination. For a long link, the delay is clearly linear in the length of the wire. The signalling rate is determined by the time to correctly sample the signal at the receiver, so the length is limited by the signal decay along the link. If there is more than one wire in the link, the signalling rate and wire length are also limited by the signal skew across the wires.

A correctly sampled analog signal can be viewed as a stream of digital symbols (phits) delivered from source to destination over time. Logical values on each wire may be conveyed by voltage levels or voltage transitions. Typically, the encoding of digital symbols is chosen so that it is easy to identify common failures, such as stuck-at faults and open connections, and easy to maintain clocking. Within the stream of symbols individual packets must be identified. Thus, part of the signaling convention of a link is its *framing*, which identifies the start and end of each packet. In a wide link, distinct control lines may identify the head and tail phits. In a narrow link, special control symbols are inserted in the stream to provide framing. In an asynchronous serial link, the clock must be extracted from the incoming analog signal as well; this is typically done with a unique synchronization burst.[PeDa96].

The Cray T3E network provides a convenient contrast to the T3D. It uses a wide, asynchronous link design. The link is 14 bits wide in each direction, operating at 375 MHz. Each "bit" is conveyed by a low voltage differential signal (LVDS) with a nominal swing of 600 mV on a pair of wires, i.e., the receiver senses the difference in the two wires, rather than the voltage relative to ground. The clock is sent along with the data. The maximum transmission distance is approximately one meter, but even at this length there will be multiple bits on the wire at a time. A flit contains five phits, so the switches operate at 75MHz on 70 bit quantities containing one 64 bit word plus control information. Flow control information is carried on data packets and idle symbols on the link in the reverse direction. The sequence of flits is framed into single-word and 8-word read and write request packets, message packets, and other special packets. The maximum data bandwidth of a link is 500 MB/s.

The encoding of the packet within the frame is interpreted by the nodes attached to the link. Typically, the envelope is interpreted by the switch to do routing and error checking. The payload is delivered uninterpreted to the destination host, at which point further layers or internal envelopes are interpreted and peeled away. However, the destination node may need to inform the source whether it was able to hold the data. This requires some kind of node-to-node information that is distinct from the actual communication, for example, an acknowledgment in the reverse direction. With wide links, control lines may run in both directions to provide this information. Thin links are almost always bidirectional, so that special flow-control signals can be inserted into the stream in the reverse direction.[1]

The Scalable Coherent Interface (SCI) defines both a long, wide copper link and a long, narrow fiber link. The links are unidirectional and nodes are always organized into rings. The copper link comprises 18 pairs of wires using differential ECL signaling on both edges of a 250 MHz clock.

---

1. This view of flow-control as inherent to the link is quite different from view in more traditional networking applications, where flow control is realized on top of the link-level protocol by special packets.

It carries 16 bits of data, clock, and a flag bit. The fiber link is serial and operates at 1.25 Gb/s. Packets are a sequence of 16-bit phits, with the header consisting of a destination node number phit and a command phit. The trailer consists of a 32-bit CRC. The flag bit provides packet framing, by distinguishing idle symbols from packet flits. At least one idle phit occurs between successive packets.

Many evaluations of networks treat links as having a fixed cost. Common sense would suggest that the cost increases with the length of the link and its width. This is actually a point of considerable debate within the field, because the relative quality of different networks depends on the cost model that is used in the evaluation. Much of the cost is in the connectors and the labor involved in attaching them, so there is a substantial fixed cost. The connector costs increases with width, whereas the wire cost increases with width and length. In many cases, the key constraint is the cross sectional area of a bundle of links, say at the bisection; this increases with width.

10.2.2 Switches

A switch consists of a set of input ports, a set of output ports, an internal "cross-bar" connecting each input to every output, internal buffering, and control logic to effect the input-output connection at each point in time, as illustrated in Figure 10-5. Usually, the number of input ports is equal to the number of output ports, which is called the *degree* of the switch.[1] Each output port includes a transmitter to drive the link. Each input port includes a matching receiver. The input port includes a synchronizer in most designs to align the incoming data with the local clock domain of the switch. This is essentially a FIFO, so it is natural to provide some degree of buffering with each input port. There may also be buffering associated with the outputs or shared buffering for the switch as a whole. The interconnect within the switch may be an actual cross bar, a set of multiplexors, a bus operating at a multiple of the link bandwidth, or an internal memory. The complexity of the control logical depends on the routing and scheduling algorithm, as we will discuss below. At the very least, it must be possible to determine the output port required by each incoming packet, and to arbitrate among input ports that need to connect to the same output port.

Many evaluations of networks treat the switch degree as its cost. This is clearly a major factor, but again there is room for debate. The cost of some parts of the switch is linear in the degree, e.g., the transmitters, receivers, and port buffers. However, the internal interconnect cost may increase with the square of the degree. The amount of internal buffering and the complexity of the routing logic also increase more than linearly with degree. With recent VLSI switches, the dominant constraint tends be the number of pins, which is proportional to the number of ports times the width of each port.

---

1. As with most rules, there are exceptions. For example, in the BBN Monarch design two distinct kinds of switches were used that had unequal number of input and output ports.[Ret*90] Switches that routed packets to output ports based on routing information in the header could have more outputs than inputs. A alternative device, called a concentrator, routed packets to any output port, which were fewer than the number of input ports and all went to the same node.
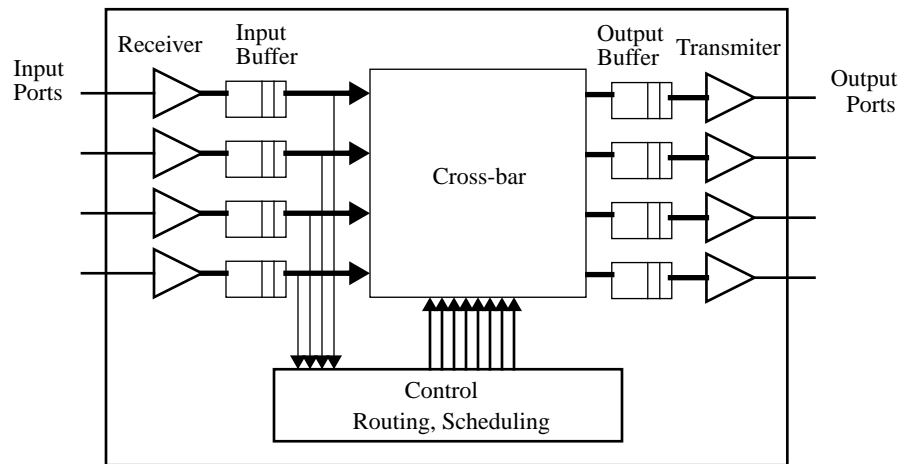
Figure 10-5 Basic Switch Organiation

A switch consists of a set of input ports, a set of output ports, an internal "cross-bar" connecting each input to every output, internal buffering, and control logic to effect the input-output connection at each point in time.

10.2.3 Network Interfaces

The network interface (NI) contains one or more input/output ports to source packets into and sink packets from the network, under direction of the communication assist which connects it to the processing node, as we have seen in previous chapters. The network interface, or host nodes, behave quite differently than switch nodes and may be connected via special links. The NI formats the packets and constructs the routing and control information. It may have substantial input and output buffering, compared to a switch. It may perform end-to-end error checking and flow control. Clearly, its cost is influenced by its storage capacity, processing complexity, and number of ports.

10.3 Interconnection Topologies

Now that we understand the basic factors determining the performance and the cost of networks, we can examine each of the major dimensions of the design space in relation to these factors. This section covers the set of important interconnection topologies. Each topology is really a class of networks scaling with the number of host nodes, N, so we want to understand the key characteristics of each class as a function of N. In practice, the topological properties, such as distance, are not entirely independent of the physical properties, such as length and width, because some topologies fundamentally require longer wires when packed into a physical volume, so it is important to understand both aspects.

10.3.1 Fully connected network

A fully-connected network consists of a single switch, which connects all inputs to all outputs. The diameter is one. The degree is N. The loss of the switch wipes out the whole network, however, loss of a link removes only one node. One such network is simply a bus, and this provides a useful reference point to describe the basic characteristics. It has the nice property that the cost scales as O(N). Unfortunately, only one data transmission can occur on a bus at once, so the total bandwidth is O(1), as is the bisection. In fact, the bandwidth scaling is worse than O(1) because the clock rate of a bus decreases with the number of ports due to RC delays. (An ethernet is really a bit-serial, distributed bus; it just operates at a low enough frequency that a large number of physical connections are possible.) Another fully connected network is a cross-bar. It provides $O(N)$ bandwidth, but the cost of the interconnect is proportional to the number of cross-points, or $O(N^2)$. In either case, a fully connected network is not scalable in practice. This is not to say they are not important. Individual switches are fully connected and provide the basic building block for larger networks. A key metric of technological advance in networks is the degree of a cost-effective switch. With increasing VLSI chip density, the number of nodes that can be fully connected by a cost-effective switch is increasing.

10.3.2 Linear arrays and rings

The simplest network is a linear array of nodes numbered consecutively $0, ..., N-1$ and connected by bidirectional links. The diameter is $N-1$, the average distance is roughly $\frac{2}{3}N$, and removal of a single link partitions the network, so the bisection width is one. Routing in such a network is trivial, since there is exactly one route between any pair of nodes. To describe the route from node A to node B, let us define $R = B - A$ to be the *relative address* of B from A. This signed, $\log N$ bit number is the number of links to cross to get from A to B with the positive direction being away from node 0. Since there is a unique route between a pair of nodes, clearly the network provides no fault tolerance. The network consists of N-1 links and can easily be laid out in $O(N)$ space using only short wires. Any contiguous segment of nodes provides a subnetwork of the same topology as the full network.

A ring or torus of N nodes can be formed by simply connecting the two ends of an array. With unidirectional links, the diameter is N-1, the average distance is N/2, the bisection width is one link, and there is one route between any pair of nodes. The relative address of B from A is $(B-A) \bmod N$. With bidirectional links, the diameter is N/2, the average distance is N/3, the degree of the node is two and the bisection is two. There are two routes (two relative addresses) between pair of nodes, so the network can function with degraded performance in the presence of a single faulty link. The network is easily laid out with $O(N)$ space using only short wires, as indicated by Figure 10-6, by simply folding the ring. The network can be partitioned into smaller subnetworks, however, the subnetworks are linear arrays, rather than rings.

Although these one dimensional networks are not scalable in any practical sense, they are an important building block, conceptually and in practice. The simple routing and low hardware complexity of rings has made them very popular for local area interconnects, including FDDI, Fiber Channel Arbitrated Loop, and Scalable Coherent Interface (SCI). Since they can be laid out with very short wires, it is possible to make the links very wide. For example, the KSR-1 used a

32-node ring as a building block which was 128 bits wide. SCI obtains its bandwidth by using 16-bit links.
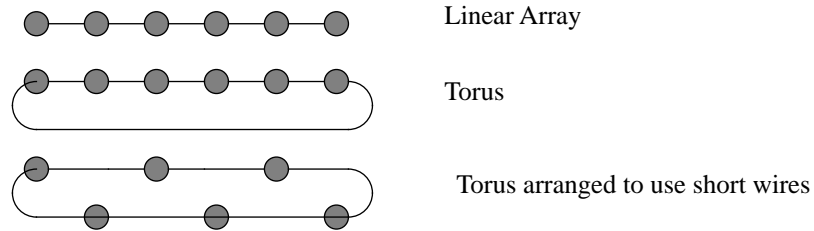

Figure 10-6 Linear and Ring Topologies

The linear array and torus are easily laid out to use uniformly short wires. The distance and cost grow a $O(N)$, whereas the aggregate bandwidth is only $O(1)$.

10.3.3 Multidimensional meshes and tori

Rings and arrays generalize naturally to higher dimensions, including 2D 'grids' and 3D 'cubes', with or without end-around connections. A d-dimensional array consists of $N = k_{d-1} \times \ldots \times k_0$ nodes, each identified by its d-vector of coordinates $(i_{d-1}, \ldots, i_0)$, where $0 \le i_j \le k_j - 1$ for $0 \le j \le d - 1$. Figure 10-6 shows the common cases of two and three dimensions. For simplicity, we will assume the length along each dimension is the same, so $N = k^d$ ($k = \sqrt[d]{N}$, $r = \log_d N$). This is called an d-dimensional k-ary array. (In practice, engineering constraints often result in non-uniform dimensions, but the theory is easily extended to handle that case.) Each node is a switch addressed by a d-vector of radix k coordinates and is connected to the nodes that differ by one in precisely one coordinate. The node degree varies between d and $2d$, inclusive, with nodes in the middle having full degree and the corners having minimal degree.

For a d dimensional k-ary torus, the edges wrap-around from each face, so every node has degree $2d$ (in the bidirectional case) and is connected to nodes differing by one (mod k) in each dimension. We will refer to arrays and tori collectively as meshes. The d-dimensional k-ary unidirectional torus is a very important class of networks, often called a k-ary d-cube, employed widely in modern parallel machines. These networks are usually employed as direct networks, so an additional switch degree is required to support the bidirectional host connection from each switch. They are generally viewed as having low degree, two or three, so the network scales by increasing k along some or all of the dimensions.

To define the routes from a node A to node B, let $R = (b_{d-1} - a_{d-1}, \ldots, b_0 - a_0)$ be the relative address of B from A. A route must cross $r_i = b_i - a_i$ links in each dimension i in the appropriate direction.The simplest approach is to traverse the dimensions in order, so for $i = 0 \ldots d - 1$ travel r_i hops in the i-th dimension. This corresponds to traveling between two locations in a metropolitan grid by driving in, say, the east-west direction, turning once, and driving in the north-south direction. Of course, we can reach the same destination by first travelling north/south and
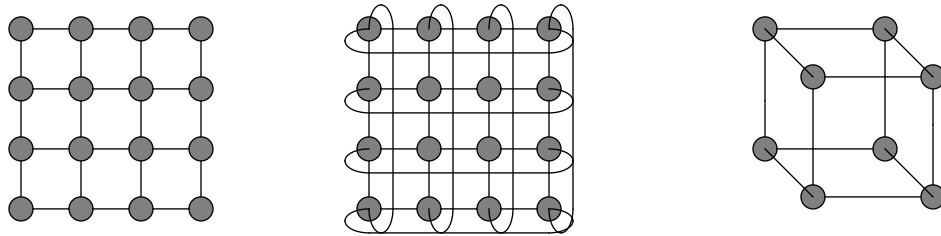
Figure 10-7 Grid, Torus, and Cube Topologies

Grids, tori, and cubes are special cases of k-ary d-cube networks, which are constructed with k nodes in each of d dimensions. Low dimensional networks pack well in physical space with short wires.

then east/west, or by zig-zagging anywhere between these routes. In general, we may view the source and destination points as corners of a subarray and follow any path between the corners that reduces the relative address from the destination at each hop.

The diameter of the network is $d(k-1)$. The average distance is simply the average distance in each dimension, roughly $d\frac{2}{3}k$. If k is even, the bisection of an d-dimensional k-ary array is k^{d-1} bidirectional links. This is obtained by simply cutting the array in the middle by a (hyper)plane perpendicular to one of the dimensions. (If k is odd, the bisection may be a little bit larger.) For a unidirectional torus, the relative address and routing generalizes along each dimension just as for a ring. All nodes have degree d (plus the host degree) and k^{d-1} links cross the middle in each direction.

It is clear that a 2-dimensional mesh can be laid out $O(N)$ space in a plane with short wires, and 3-dimensional mesh in $O(N)$ volume in free space. In practice, engineering factors come into play. It is not really practical to build a huge 2D structure and mechanical issues arise in how 3D is utilized, as illustrated by the following example.

Example 10-1   Using a direct 2D array topology, such as in the Intel Paragon where a single cabinet holds 64 processors forming a 4 wide by 16 high array of nodes (each node containing a message processor and 1 to 4 compute processors), how might you configure cabinets to construct a large machine with only short wires?

Answer   Although there are many possible approaches, the one used by Intel is illustrated in Figure 1-24 of Chapter 1. The cabinets stand on the floor and large configurations are formed by attaching these cabinets side by side, forming a $16 \times k$ array. The largest configuration was a 1824 node machine at Sandia National Laboratory

configured as a 16×114 array. The bisection bandwidth is determined by the 16 links that cross between cabinets.

Other machines have found alternative strategies for dealing with real world packaging restrictions. The MIT J-machine is a 3D torus, where each board comprises an 8x16 torus in the first two dimensions. Larger machines are constructed by stacking these boards next to one another, with board-to-board connections providing the links in the third dimension. The Intel ASCI Red machine with 4536 compute nodes is constructed as 85 cabinets. It allows long wires to run between cabinets in each dimension. In general, with higher dimensional meshes several logical dimensions are embedded in each physical dimension using longer wires. Figure 10-6 shows a 6x3x2 array and a 4-dimensional 3-ary array embedded in a plane. It is clear that for a given physical dimension, the average wire length and the number of wires increases with the number of logical dimensions.
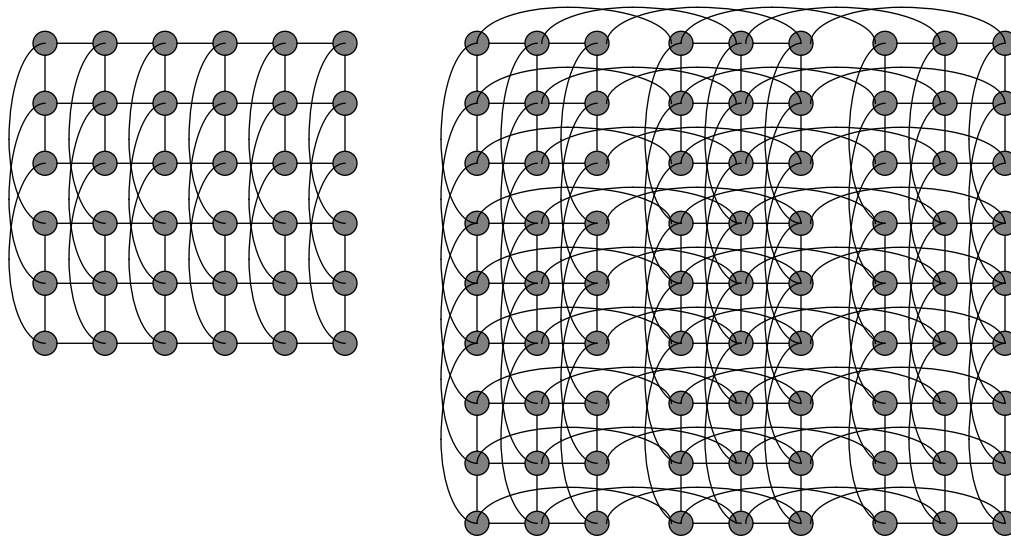
Figure 10-8 Embeddings of many logical dimensions in two phjysical dimensions

Higher dimensional k-ary d-cube can be laid out in 2D by replicating a 2D slice and then connecting slices across the remaining dimensions. This wiring complexity of these higher dimensional networks can be easily seen in the figure.

10.3.4 Trees

In meshes, the diameter and average distance increases with the d-th root of N. There are many other topologies where the routing distance grows only logarithmically. The simplest of these is a tree. A binary tree has degree three.Typically, trees are employed as indirect networks with hosts as the leaves, so for N leaves the diameter is $2\log N$. (Such a topology could be used as a direct network of $N = k\log k$ nodes.) In the indirect case, we may treat the binary address of each node as a $d = \log N$ bit vector specifying a path from the root of the tree – the high order bit indicates whether the node is below the left or right child of the root, and so on down the levels of the tree.

The levels of the tree correspond directly to the "dimension" of the network. One way to route from node A to node B would be to go all the way up to the root and then follow the path down specified by the address of B. Of course, we really only need to go up to the first common parent of the two nodes, before heading down. Let $R = B \otimes A$, the bit-wise xor of the node addresses, be the relative address of A B and i be the position of the least significant 1 in R. The route from A to node B is simply i hops up and the low-order bits of B down.

Formally, a complete indirect binary tree is a network of $2N-1$ nodes organized as $d = \log_2 N$ levels. Host nodes occupy level 0 and are identified by a d-bit level address $A = a_{d-1}, ..., a_0$. A switch node is identified by its level, i, and its $d-i$ bit level address, $A^{(i)} = a_{d-1}, ..., a_i$. A switch node $[i, A^{(i)}]$ is connected to a parent, $[i+1, A^{(i+1)}]$, and two children, $[i-1, A^{(i)} \| 0]$ and $[i-1, A^{(i)} \| 1]$, where the vertical bars indicate bit-wise concatenation. There is a unique route between any pair of nodes by going up to the least common ancestor, so there is no fault tolerance. The average distance is almost as large as the diameter and the tree partitions into sub-trees. One virtue of the tree is the ease of supporting broadcast or multicast operations

Clearly, by increasing the branching factor of the tree the routing distance is reduced. In a k-ary tree, each node has k children, the height of the tree is $d = \log_k N$ and the address of a host is specified by a d-vector of radix k coordinates describing the path down from the root.

One potential problem with trees is that they seem to require long wires. After all, when one draws a tree in a plane, the lines near the root appear to grow exponentially in length with the number of levels, as illustrated in the top portion of Figure 10-9, resulting in a $O(N\log N)$ layout with $O(N)$ long wires. This is really a matter of how you look at it. The same 16-node tree is laid out compactly in two dimensions using a recursive "H-tree" pattern, which allows an $O(N)$ layout with only $O\sqrt{N}$ long wires [BhLe84]. One can imagine using the H-tree pattern with multiple nodes on a chip, among nodes (or subtrees) on a board, or between cabinets on a floor, but the linear layout might be used between boards.

The more serious problem with the tree is its bisection. Removing a single link near the root bisects the network. It has been observed that computer scientists have a funny notion of trees; real trees get thicker toward the trunk. An even better analogy is your circulatory system, where the heart forms the root and the cells the leaves. Blood cells are routed up the veins to the root and down the arteries to the cells. There is essentially constant bandwidth across each level, so that blood flows evenly. This idea has been addressed in an interesting variant of tree networks, called fat-trees, where the upward link to the parent has twice the bandwidth of the child links. Of course, packets don't behave quite like blood cells, so there are some issues to be sorted out on how exactly to wire this up. These will fall out easily from butterflies.

10.3.5 Butterflies

The constriction at the root of the tree can be avoided if there are "lots of roots." This is provided by an important logarithmic network, called a butterfly. (The butterfly topology arises in many settings in the literature. It is the inherent communication pattern on an element-by-element level of the FFT, the Batcher odd-even merge sort, and other important parallel algorithms. It is isomorphic to topologies in the networking literature including the Omega and SW-Banyan net-
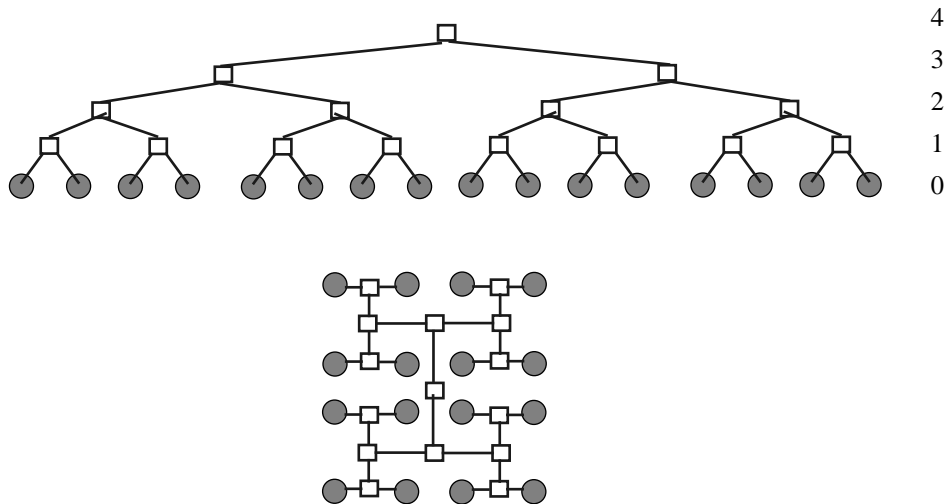
Figure 10-9 Binary Trees

Trees provide a simple network with logarithmic depth. They can be laid out efficiently in 2D by using an H-tree configuration. Routing is simple and they contain trees as subnetworks. However, the bisection bandwidth is only $O(1)$.

works, and closely related to the shuffle-exchange network and the hypercube, which we discuss later.) Given 2x2 switches, the basic building-block of the butterfly obtained by simply crossing one of each pair of edges, as illustrated in the top of Figure 10-10. This is a tool for correcting one bit of the relative address – going straight leaves the bit the same, crossing flips the bit. These 2x2 butterflies are composed into a network of $N = 2^d$ nodes in $\log_2 N$ levels by systematically changing the cross edges. A 16-node butterfly is illustrated in the bottom portion of Figure 10-10. This configuration shows an indirect network with unidirectional links going upward, so hosts deliver packets into level 0 and receive packets from level d. Each level corrects one additional bit of the relative address. Each node at level d forms the root of a tree with all the hosts as leaves., and from each host there is a tree of routes reaching every node at level d.

A d-dimensional indirect butterfly has $N = 2^d$ host nodes and $d2^{d-1}$ switch nodes of degree 2 organized as d levels of $\frac{N}{2}$ nodes each. A switch node at level i, $[i, A]$ has its outputs connected to nodes $[i+1, A]$ and $[i+1, A \otimes 2^i]$. To route from A to B, compute the relative address $R = A \otimes B$ and at level i use the "straight edge" if r_i is 0 and the cross edge otherwise. The diameter is $\log N$. In fact, all routes are $\log N$ long. The bisection is $\frac{N}{2}$. (A slightly different formulation with only one host connected to each edge switch has bisection N, but twice as many switches at each level and one additional level.)
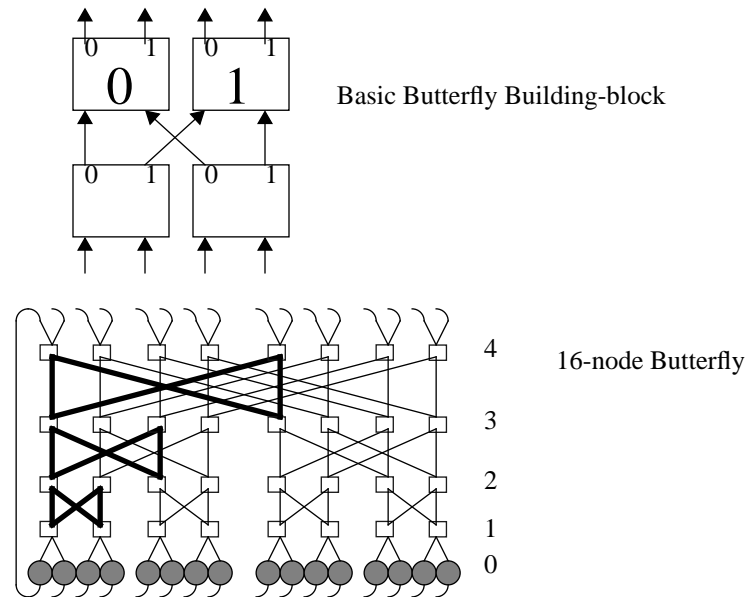
Figure 10-10 Butterfly

Butterflies are a logarithmic depth network constructed by composing 2x2 blocks which correct one bit in the relative address. It can be viewed as a tree with multiple roots.

A d-dimensional k-ary butterfly is obtained using switches of degree k, a power of 2. The address of a node is then viewed as a d-vector of radix k coordinates, so each level corrects $\log k$ bits in the relative address. In this case, there are $\log_k N$ levels. In effect, this fuses adjacent levels into a higher radix butterfly.

There is exactly one route from each host output to each host input, so there is no inherent fault tolerance in the basic topology. However, unlike the 1D mesh or the tree where a broken link partitions the network, in the butterfly there is the potential for fault tolerance. For example, the route from A to B may be broken, but there is a path from A to another node C and from C to B. There are may proposals for making the butterfly fault tolerant by just adding a few extra links. One simple approach is add an extra level to the butterfly so there are two routes to every destination from every source. This approach was used in the BBN T2000.

The butterfly appears to be a qualitatively more scalable network than meshes and trees, because each packet crosses $\log N$ links and there are $N \log N$ links in the network, so on average it should be possible for all the nodes to send messages anywhere all at once. By contrast, on a 2D torus or tree there are only two links per node, so nodes can only send messages a long distance infrequently and very few nodes are close. A similar argument can be made in terms of bisection. For a random permutation of data among the N nodes, $\frac{N}{2}$ messages are expected to cross the bisection in each direction. The butterfly has $\frac{N}{2}$ links across the bisection, whereas the d-dimensional

mesh has only $N^{\frac{d-1}{d}}$ links and the tree only one. Thus, as the machine scales, a given node in a butterfly can send every other message to a node on the other side of the machine, whereas a node in a 2D mesh can send only every $\frac{\sqrt[d]{N}}{2}$ -th message and the tree only every N -th message to the other side.

There are two potential problems with this analysis, however. The first is cost. In a tree or d -dimensional mesh, the cost of the network is a fixed fraction of the cost of the machine. For each host node there is one switch and d links. In the butterfly the cost of the network per node increases with the number of nodes, since for each host node there is $\log N$ switches and links. Thus, neither scales perfectly. The real question comes down to what a switch costs relative to a processor and what fraction of the overall cost of the machine we are willing to invest in the network in order to be able to deliver a certain communication performance. If the switch and link is 10% of the cost of the node, then on a thousand processor machine the network will be only one half of the total cost with a butterfly. On the other hand, if the switch is equal in cost to a node, we are unlikely to consider more than a low dimensional network. Conversely, if we reduce the dimension of the network, we may be able to invest more in each switch.

The second problem is that even though there is enough links in the butterfly to support bandwidth-distance product of a random permutation, the topology of the butterfly will not allow an arbitrary permutation of N message among the N nodes to be routed without conflict. A path from an input to an output blocks the paths of many other input-output pairs, because there are shared edges. In fact, even if going through the butterfly twice, there are permutations that cannot be routed without conflicts. However, if two butterflies are laid back-to-back, so a message goes through one forward and the other in the reverse direction, then for any permutation there is a choice of intermediate positions that allow a conflict-free routing of the permutation. This back-to-back butterfly is called a Benes[Ben65,Lei92] network and it has been extensively studied because of its elegant theoretical properties. It is often seen as having little practical significance because it is costly to compute the intermediate positions and the permutation has to be known in advance. On the other hand, there is another interesting theoretical result that says that on a butterfly any permutation can be routed with very few conflicts with high probability by first sending every message to a random intermediate node and then routing them to the desired destination[Lei92]. These two results come together in a very nice practical way in the fat-tree network.

A d -dimensional k -ary fat-tree is formed by taking a d -dimensional k -ary Benes network and folding it back on itself at the high order dimension, as illustrated in Figure 10-10. The collection of $\frac{N}{2}$ switches at level i are viewed as N^{d-i} "fat nodes" of 2^{i-1} switches. The edges of the forward going butterfly go up the tree toward the roots and the edges of the reverse butterfly go down toward the leaves. To route from A to B, pick a random node C in the least common ancestor fat-node of A to B, take the unique tree route from A to C and the unique tree route back down from C to B. Let i be the highest dimension of difference in A and B, then there are 2^i root nodes to choose from, so the longer the routing distance the more the traffic can be distributed. This topology clearly has a great deal of fault tolerance, it has the bisection of the butterfly, the partitioning properties of the tree, and allows essentially all permutations to be routed with very little contention. It is used in the Connection Machine CM-5 and the Meiko CS-2. In

the CM-5 the randomization on the upward path is down dynamically by the switches, in the CS-2 the source node chooses the ancestor. A particularly important practical property of butterflies and fat-trees is that the nodes have fixed degree, independent of the size of the network. This allows networks of any size to be constructed with the same switches. As is indicated in Figure 10-11, the physical wiring complexity of the higher levels of a fat-tree or any butterfly-like network become critical, as there are a large number of long wires that connect to different places.

16-node Benes Network (Unidirectional)
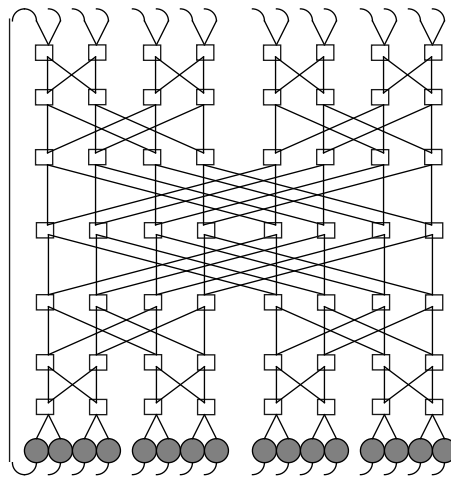
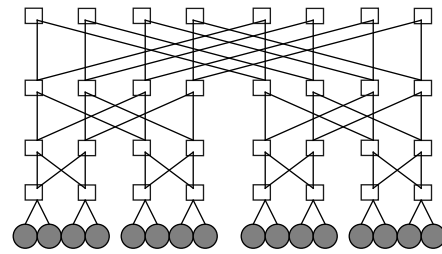

16-node 2-ary Fat-Tree (Bidirectional)

Figure 10-11 Benes network and fat-tree

A Benes network is constructed essentially by connecting two butterflies back-to-back. It has the interesting property that it can route any permutation in a conflict free fashion, given the opportunity to compute the route off-line. Two forward going butterflies do not have this property. The fat tree is obtained by folding the second half of the Benes network back on itself and fusing the two directions, so it is possible to turn around at each level.

10.3.6 Hypercubes

It may seem that the straight-edges in the butterfly are a target for potential, since they take a packet forward to the next level in the same column. Consider what happens if we collapse all the switches in a column into a single $\log N$ degree switch. This has brought us full circle; it is exactly a d-dimensional 2-ary torus! It is often called a hypercube or binary n-cube. Each of the $N = 2^d$ nodes are connected to the d nodes that differ by exactly one bit in address. The relative address $R(A, B) = A \otimes B$ specifies the dimensions that must be crossed to go from A to B. Clearly, the length of the route is equal to the number of ones in the relative address. The dimensions can be corrected in any order (corresponding the different ways of getting between opposite corners of the subcube) and the butterfly routing corresponds exactly to dimension order routing, called *e-cube routing* in the hypercube literature. Fat-tree routing corresponds to picking a random node in the subcube defined by the high order bit in the relative address, sending the packet "up" the random node and back "down" to the destination. Observe that the fat-tree uses distinct sets of links for the two directions, so to get the same properties we need a pair of bidirectional links between nodes in the hypercube.

The hypercube is an important topology that has received tremendous attention in the theoretical literature. For example, lower dimensional meshes can be embedded one-to-one in the hypercube by choosing an appropriate labeling of the nodes. Recall, a greycode sequence orders the numbers from 0 to $2^d - 1$ so adjacent numbers differ by one bit. This shows how to embed a 1D mesh into a d-cube, and it can be extended to any number of dimensions. (See exercises) Clearly, butterflies, shuffle-exchange networks and the like embed easily. Thus, there is a body of literature on "hypercube algorithms" which subsumes the mesh and butterfly algorithms. (Interestingly, a d-cube does not quite embed a $d - 1$ level tree, because there is one extra node in the d-cube.

Practically speaking, the hypercube was used by many of the early large scale parallel machines, including the Cal Tech research prototypes[Seit85], the first three Intel iPSC generations[Rat85], and three generations of nCUBE machines. Later large scale machines, including the Intel Delta Intel Paragon, and Cray T3D have used low dimensional meshes. One of reasons for the shift is that in practice the hypercube topology forces the designer to use switches of a degree that supports the largest possible configuration. Ports are wasted in smaller configuration. The k-ary d-cube approach provides the practical scalability of allowing arbitrary sized configurations to be construct with a given set of components, i.e., with switches of fixed degree. None the less, this begs the question of what should be the degree?

The general trend in network design in parallel machines is toward switches that can be wired in an arbitrary topology. This what we see, for example, in the IBM SP-2, SGI Origin, Myricom network, and most ATM switches. The designer may choose to adopt a particular regular toplogy, or may wire together configurations of different sizes differently. At any point in time, technology factors such as pin-out and chip area limit the largest potential degree. Still, it is a key question, how large should be the degree?

10.4 Evaluating Design Trade-offs in Network Topology

The k-ary d-cube provides a convenient framework for evaluating desing alternatives for direct networks. The design question can be posed in two ways. Given a choice of dimension, the design of the switch is determined and we can ask how the machine scales. Alternatively, for the machine scale of interest, i.e., $N = k^d$, we may ask what is the best dimensionality under salient cost constraints. We have the 2D torus at one extreme, the hypercube at the other, and a spectrum of networks between. As with most aspects of architecture, the key to evaluating trade-offs is to define the cost model and performance model and then to optimize the design accordingly. Network topology has been a point of lively debate over the history of parallel architectures. To a large extent this is because different positions make sense under different cost models, and the technology keeps changing. Once the dimensionality (or degree) of the switch is determined the space of candidate networks is relatively constrained, so the question that is being addressed is how large a degree is worth working toward.

Let's collect what we know about this class of networks in one place. The total number of switches is N, regardless of degree, however, the switch degree is d so the total number of links is $C = Nd$ and there are $2wd$ pins per node. The average routing distance is $d\left(\frac{k-1}{2}\right)$, the diam-

eter is $d(k-1)$, and $k^{d-1} = \dfrac{N}{k}$ links cross the bisection in each direction (for even k). Thus, there are $\dfrac{2Nw}{k}$ wires crossing the middle of the network.

If our primary concern is the routing distance, then we are inclined to maximize the dimension and build a hypercube. This would be the case with store-and-forward routing, assuming that the degree of the switch and the number of links were not a significant cost factor. In addition, we get to enjoy its elegant mathematical properties. Not surprisingly, this was the topology of choice for most of the first generation large-scale parallel machine. However, with cut-through routing and a more realistic hardware cost model, the choice is much less clear. If the number of links or the switch degree are the dominant costs, we are inclined to minimize the dimension and build a mesh. For the evaluation to make sense, we want to compare the performance of design alternatives with roughly equal cost. Different assumptions about what aspects of the system are costly lead to very different conclusions.

The assumed communication pattern influences the decision too. If we look at the worst case traffic pattern for each network, we will prefer high dimensional networks where essentially all the paths are short. If we look at patterns where each node is communicating with only one or two near neighbors, we will prefer low dimensional networks, since only a few of the dimensions are actually used.

10.4.1 Unloaded Latency

Figure 10-12 shows the increase in unloaded latency under our model of cut-through routing for 2, 3, and 4-cubes, as well as binary d-cubes ($k = 2$), as the machine size is scaled up. It assumes unit routing delay per stage and shows message sizes of 40 and 140 flits. This reflects two common message sizes, for $w = 1$ byte. The bottom line shows the portion of the latency resulting from channel occupancy. As we should expect, for smaller messages (or larger routing delay per stage) the scaling of the low dimension networks is worse because a message experiences more routing steps on average. However, in making comparisons across the curves in this figure, we are tacitly assuming that the difference is degree is not a signficant component of the system cost. Also, one cycle routing is very aggressive; more typical values for high performance switches are 4-8 network cycles (See Table 10-1). On the other hand, larger message sizes are also common.

To focus our attention on the dimensionality of the network as a design issue, we can fix the cost model and the number of nodes that reflects our design point and examine the performance characteristics of networks with fixed cost for a range of d. Figure 10-13 shows the unloaded latency for short messages as a function of the dimensionality for four machine sizes. For large machines the routing delays in low dimensionality networks dominate. For higher dimensionality, the latency approaches the channel time. This "equal number of nodes" cost model has been widely used to support the view that low dimensional networks do not scale well.

It is not surprising that higher dimensional network are superior under the cost model of Figure 10-13, since the added switch degree, number of channels and channel length comes essentially for free. The high dimension networks have a much larger number of wires and pins, and bigger switches than the low dimension networks. For the rightmost end of the graph in Figure 10-13, the network design is quite impractical. The cost of the network is a significant fraction of the cost of a large scale parallel machine, so it makes sense to compare equal cost
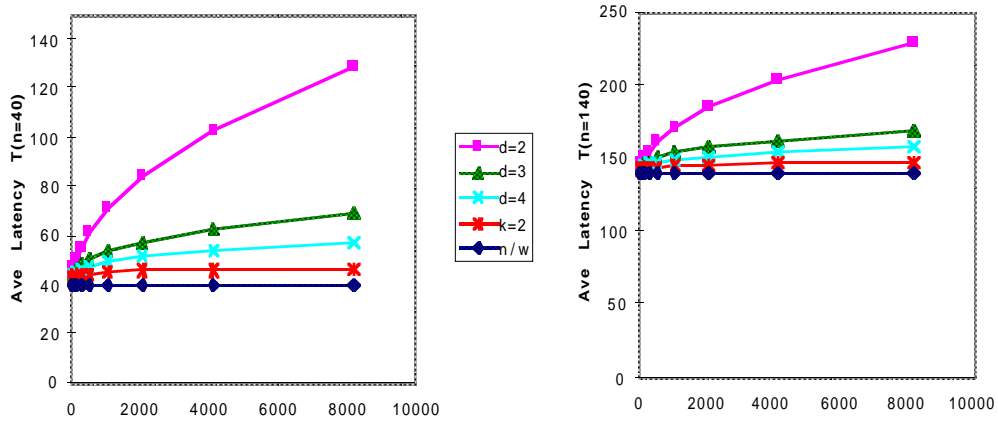
Figure 10-12 Unloaded Latency Scaling of networks for various dimensions with fixed link width

The n/w line shows the channel occupancy component of message transmission, i.e, the time for the bits to cross a single channel, which is independent of network topology. The curves show the additional latency due to routing
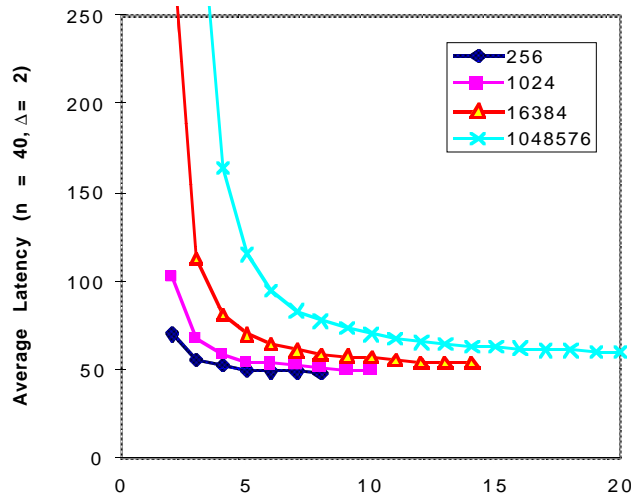
Figure 10-13 Unloaded Latency for k-ary d-cubes with Equal Node Count (n=14B, Δ = 2) as a function of the degree

With the link width and routing delay fixed, the unloaded latency for large networks rises sharply at low dimensions due to routing distance.

designs under appropriate technological assumptions. As chip size and density improves, the switch internals tend to become a less significant cost factor, whereas pins and wires remain critical. In the limit, the physical volume of the wires presents a fundamental limit on the amount of

interconnection that is possible. So lets compare these networks under assumptions of equal wiring complexity.

One sensible comparison is to keep the total number of wires per node constant, i.e., fix the number of pins $2dw$. Let's take as our baseline a 2-cube with channel width $w = 32$, so there are a total of 128 wires per node. With more dimensions there are more channels and they must each be thinner. In particular, $w_d = \lfloor 64/d \rfloor$. So, in an 8-cube the links are only 8 bits wide. Assuming 40 and 140 byte messages, a uniform routing delay of 2 cycles per hop, and uniform cycle time, the unloaded latency under equal pin scaling is shown in Figure 10-14. This figure shows a very different story. As a result of narrower channels, the channel time becomes greater with increasing dimension; this mitigates the reduction in routing delay stemming from the smaller routing distance. The very large configurations still experience large routing delays for low dimensions, regardless of the channel width, but all the configurations have an optimum unloaded latency at modest dimension.
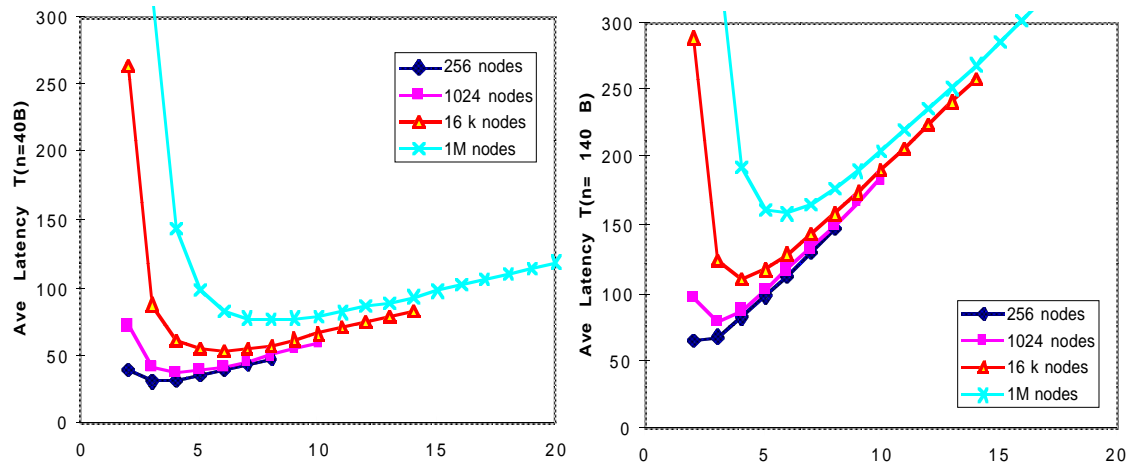
Figure 10-14 Unloaded Latency for k-ary d-cubes with Equal Pin Count (n=40B and n=140B, $\Delta = 2$)

With equal pin count, higher dimensions implies narrower channels, so the optimal design point balances the routing delay, which increases with lower dimension, against channel time, which increases with higher dimension.

If the design is not limited by pin count, the critical aspect of the wiring complexity is likely to be the number of wires that cross through the middle of the machine. If the machine is viewed as laid out in a plane, the physical bisection width grows only with the square root of the area, and in three-space it only grows as the two-thirds power of the volume. Even if the network has a high logical dimension, it must be embedded in a small number of physical dimensions, so the designer must contend with the cross-sectional area of the wires crossing the midplane.

We can focus on this aspect of the cost by comparing designs with equal number of wires crossing the bisection. At one extreme, the hypercube has N such links. Let us assume these have unit

size. A 2D torus has only $2\sqrt{N}$ links crossing the bisection, so each link could be $\frac{\sqrt{N}}{2}$ times the width of that used in the hypercube. By the equal bisection criteria, we should compare a 1024 node hypercube with bit-serial links with a torus of the same size using 32-bit links. In general, the d-dimensional mesh with the same bisection width as the N node hypercube has links of width $w_d = \frac{\sqrt[d]{N}}{2} = \frac{k}{2}$. Assuming cut-through routing, the average latency of an n-byte packet to a random destination on an unloaded network is as follows.

$$T(n, N, d) = \frac{n}{w_d} + \Delta \cdot d\left(\frac{k-1}{2}\right) = \frac{n}{k/2} + \Delta \cdot d\left(\frac{k-1}{2}\right) = \frac{n}{\sqrt[d]{N}/2} + \Delta \cdot d\left(\frac{\sqrt[d]{N}-1}{2}\right) \qquad \textbf{(EQ 10.7)}$$

Thus, increasing the dimension tends to decrease the routing delay but increase the channel time, as with equal pin count scaling. (The reader can verify that the minimum latency is achieved when the two terms are essentially equal.) Figure 10-15 shows the average latency for 40 byte messages, assuming $\Delta = 2$, as a function of the dimension for a range of machine sizes. As the dimension increases from $d = 2$ the routing delay drops rapidly whereas the channel time increases steadily throughout as the links get thinner. (The $d = 2$ point is not shown for $N = 1M$ nodes because it is rather ridiculous; the links are 512 bits wide and the average number of hops is 1023.) For machines up to a few thousand nodes, $\sqrt[3]{N}/2$ and $\log N$ are very close, so the impact of the additional channels on channel width becomes the dominant effect. If large messages are considered, the routing component becomes even less significant. For large machines the low dimensional meshes under this scaling rule become impractical because the links become very wide.

Thus far we have concerned ourselves with the wiring cross-sectional area, but we have not worried about the wire length. If a d-cube is embedded in a plane, i.e., $d/2$ dimensions are embedded in each physical dimension, such that the distance between the centers of the nodes is fixed, then each additional dimension increases the length of the longest wire by a $\sqrt{k}$ factor. Thus, the length of the longest wire in a d-cube is $k^{\frac{n}{2}-1}$ times that in the 2-cube. Accounting for increased wire length further strengthens the argument for a modest number of dimensions. There are three ways this accounting might be done. If one assumes that multiple bits are pipelined on the wire, then the increased length effectively increases the routing delay. If the wires are not pipelined, then the cycle time of the network is increased as a result of the long time to drive the wire. If the drivers are scaled up perfectly to meet the additional capacitive load, the increase is logarithmic in the wire length. If the signal must propagate down the wire like a transmission line, the increase is linear in the length.

The embedding of the d-dimensional network into few physical dimensions introduces second-order effects that may enhance the benefit of low dimensions. If a high dimension network is embedded systematically into a plane, the wire density tends to be highest near the bisection and low near the perimeter. A 2D mesh has uniform wire density throughout, so it makes better use of the area that it occupies.
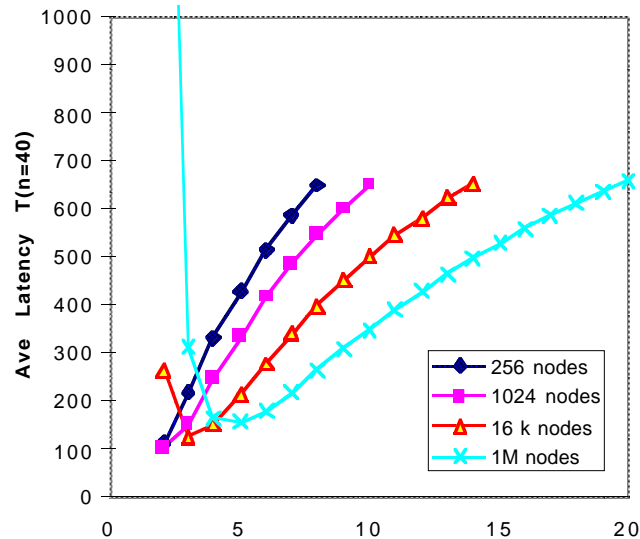
Figure 10-15 Unloaded Latency for k-ary d-cubes Equal Bisection Width (n=40B, Δ=2).

The balance between routing delay and channel time shifts even more in favor of low degree networks with an equal bisection width scaling rule.

We should also look at the trade-offs in network design from a bandwidth viewpoint. The key factor influencing latency with equal wire complexity scaling is the increased channel bandwidth at low dimension. Wider channels are beneficial if most of the traffic arises from or is delivered to one or a few nodes. If traffic is localized, so each node communicates with just a few neighbors, only a few of the dimensions are utilized and again the higher link bandwidth dominates. If a large number of nodes are communicating throughout the machine, then we need to model the effects of contention on the observed latency and see where the network saturates.

Before leaving the examination of trade-offs for latency in the unloaded case, we note that the evaluation is rather sensitive to the relative time to cross a wire and to cross a switch. If the routing delay per switch is 20 times that of the wire, the picture is very different, as shown in Figure 10-16.

10.4.2 Latency under load

In order to analyze the behavior of a network under load, we need to capture the effects of traffic congestion on all the other traffic that is moving through the network. These effects can be subtle and far reaching. Notice when you are next driving down a loaded freeway, for example, that where a pair of freeways merge and then split the traffic congestion is worse than even a series of on-ramps merging into a single freeway. There is far more driver-to-driver interaction in the interchange and at some level of traffic load the whole thing just seems to stop. Networks behave in a similar fashion, but there are many more interchanges. (Fortunately, rubber-necking at a roadside
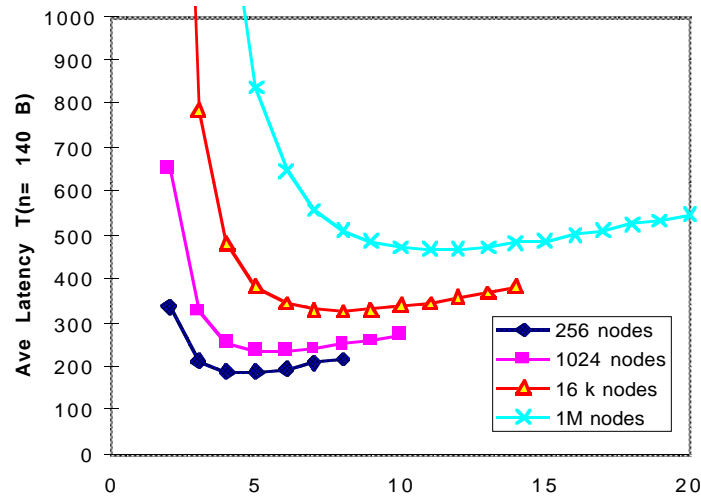
Figure 10-16 Unloaded Latency for k-ary d-cubes with Equal Pin Count and larger routing delays(n=140B, $\Delta = 20$)

When the time to cross a switch is significantly larger than the time to cross a wire, as is common in practice, higher degree switches become much more attractive.

disturbance is less of an issue). In order to evaluate these effects in a family of topologies, we must take a position on the traffic pattern, the routing algorithm, the flow-control strategy, and a number of detailed aspects of the internal design of the switch. One can then either develop a queuing model for the system or build a simulator for the proposed set of designs. The trick, as in most other aspects of computer design, is to develop models that are simple enough to provide intuition at the appropriate level of design, yet accurate enough to offer useful guidance as the design refinement progresses.

We will use a closed-form model of contention delays in k-ary d-cubes for random traffic using dimension-order cut-through routing and unbounded internal buffers, so flow control and deadlock issues do not arise, due to Agarwal [Aga91]. The model predictions correlate well with simulation results for networks meeting the same assumptions. This model is based on earlier work by Kruskal and Snir modeling the performance of indirect (Banyan) networks [KrSn83]. Without going through the derivation, the main result is that we can model the latency for random communication of messages of size n on a k-ary d-cube with channel width w at a load corresponding to an aggregate channel utilization of ρ by:

$$T(n, k, d, w, \rho) = \frac{n}{w} + h_{\text{ave}}(\Delta + W(n, k, d, w, \rho))$$

$$W(n, k, d, w, \rho) = \frac{n}{w} \cdot \frac{\rho}{1 - \rho} \cdot \frac{h_{\text{ave}} - 1}{h_{\text{ave}}^2} \cdot \left(1 + \frac{1}{d}\right)$$

(EQ 10.8)

where $h_{\text{ave}} = d\left(\frac{k - 1}{2}\right)$.

Using this model, we can compare the latency under load for networks of low or high dimension of various sizes, as we did for unloaded latency. Figure 10-17 shows the predicted latency on a 1000 node 10-ary 3-cube and a 1024 node 32-ary 2-cube as a function of the requested aggregate channel utilization, assuming equal channel width, for relatively small messages sizes of 4, 8, 16, and 40 phits. We can see from the right-hand end of the curves that the two networks saturate at roughly the same channel utilization, however, this saturation point decreases rapidly with the message size. The left end of the curves indicates the unloaded latency. The higher degree switch enjoys a lower base routing delay with the same channel time, since there are fewer hops and equal channel widths. As the load increases, this difference becomes less significant. Notice how large is the contended latency compared to the unloaded latency. Clearly, in order to deliver low latency communication to the user program, it is important that the machine is designed so that the network does not go into saturation easily; either by providing excess network bandwidth, "headroom", or by conditioning the processor load.
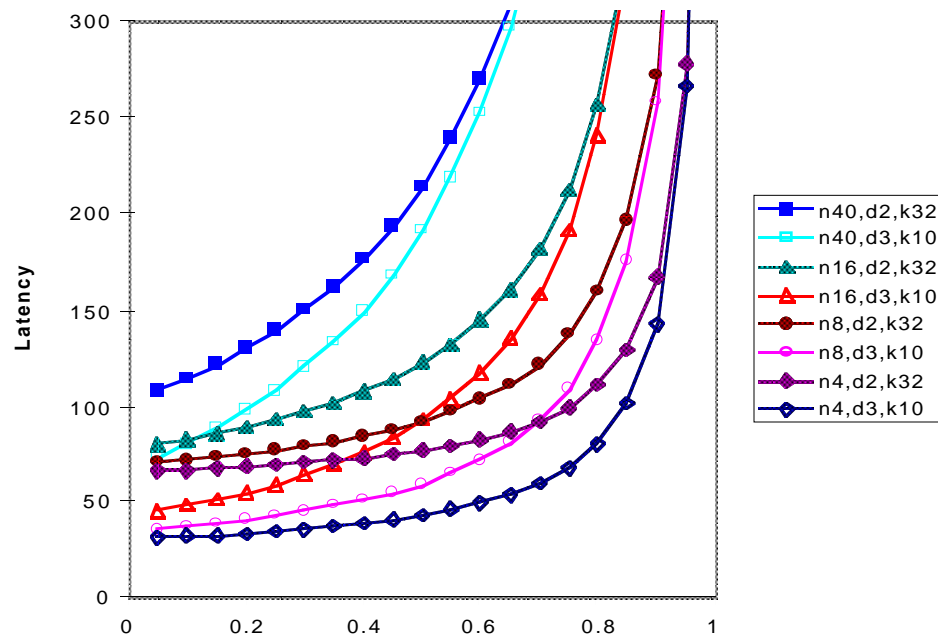
Figure 10-17 Latency with Contention vs. load for 2-ary 32-cube and 3-ary 10-cube with routing delay 2

At low channel utilization the higher dimensional network has significantly low latency, with equal channel width, but as the utilization increases they converge toward the same saturation point.

The data in Figure 10-17 raises a basic tradeoff in network design. How large a packet should the network be designed for? The data shows clearly that networks move small packets more efficiently than large ones. However, smaller packets have worse packet efficiency, due to the routing and control information contained in each one, and require more network interface events for the same amount of data transfer.

We must be careful about the conclusions we draw from Figure 10-17 regarding the choice of network dimension. The curves in the figure show how efficiently each network utilizes the set of channels it has available to it. The observation is that both use their channels with roughly equal effectiveness. However, the higher dimensional network has a much greater available bandwidth per node; it has 1.5 times as many channels per node and each message uses fewer channels. In a k-ary d-cube the available flits per cycle under random communication is $\dfrac{Nd}{d\dfrac{(k-1)}{2}}$, or

$2/(k-1)$ flits per cycle per node ($\dfrac{2w}{k-1}$ bits per cycle). The 3-cube in our example has almost four times as much available bandwidth at the same channel utilization, assuming equal channel width. Thus, if we look at latency against delivered bandwidth, the picture looks rather different, as shown in Figure 10-18. The 2-cube starts out with a higher base latency and saturates before the 3-cube begins to feel the load.
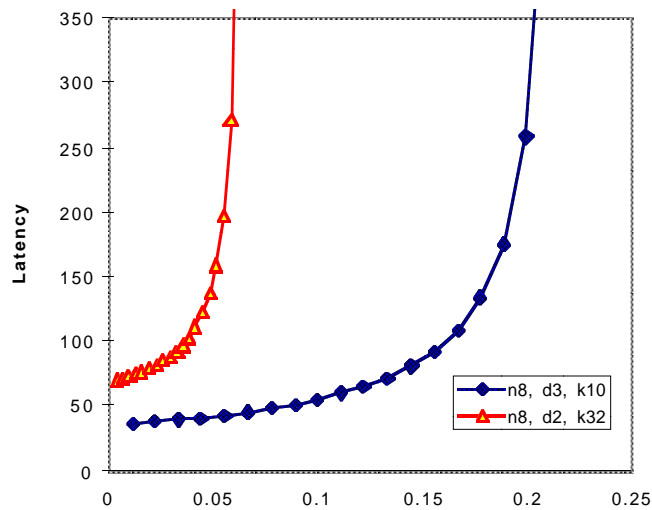
Figure 10-18 Latency vs flits per cycle with Contention for 2-ary 32-cube and 3-ary 10-cube. Routing delay 2

This comparison brings us back to the question of appropriate equal cost comparison. As an exercise, the reader can investigate the curves for equal pin out and equal bisection comparison. Widening the channels shifts the base latency down by reducing channel time, increases the total bandwidth available, and reduces the waiting time at each switch, since each packet is serviced faster. Thus, the results are quite sensitive to the cost model.

There are interesting observations that arise when the width is scaled against dimension. For example, using the equal bisection rule, the capacity per node is. $C(N, d) = w_d \cdot \dfrac{2}{k-1} \approx 1$. The aggregate capacity for random traffic is essentially independent of dimension! Each host can

expect to drive, on average, a fraction of a bit per network clock period. This observation yields a new perspective on low dimension networks. Generally, the concern is that each of several nodes must route messages a considerable distance along one dimension. Thus, each node must send packets infrequently. Under the fixed bisection width assumption, with low dimension the channel becomes a shared resource pool for several nodes, whereas high dimension networks partition the bandwidth resource for traffic in each of several dimensions. When a node uses a channel, in the low dimension case it uses it for a shorter amount of time. In most systems, pooling results in better utilization than partitioning.

In current machines, the area occupied by a node is much larger than the cross-section of the wires and the scale is generally limited to a few thousand nodes. In this regime, the bisection of the machine is often realized by bundles of cables. This does represent a significant engineering challenge, but it is not a fundamental limit on the machine design. There is a tendency to use wider links and faster signalling in topologies with short wires, as illustrated by Table 10-1, but it is not as dramatic as the equal bisection scaling rule would suggest.

10.5 Routing

The routing algorithm of a network determines which of the possible paths from source to destination are used as routes and how the route followed by each particular packet is determined. We have seen, for example, that in a k-ary d-cube the set of shortest routes is completely described by the relative address of the source and destination, which specifies the number of links that need to be crossed in each dimension. Dimension order routing restricts the set of legal paths so that there is exactly one route from each source to each destination, the one obtained by first traveling the correct distance in the high-order dimension, then the next dimension and so on. This section describes the different classes of routing algorithms that are used in modern machines and the key properties of good routing algorithms, such as producing a set of deadlock-free routes, maintaining low latency, spreading load evenly, and tolerating faults.

10.5.1 Routing Mechanisms

Let's start with the nuts and bolts. Recall, the basic operation of a switch is to monitor the packets arriving at its inputs and for each input packet select an output port on which to send it out. Thus, a routing algorithm is a function $R : N \times N \to C$, which at each switch maps the destination node, n_d, to the next channel on the route. There are basically three mechanisms that high-speed switches use to determine the output channel from information in the packet header: arithmetic, source-based port select, and table look-up. All of these mechanisms are drastically simpler than the kind of general routing computations performed in traditional LAN and WAN routers. In parallel computer networks, the switch needs to be able to make the routing decision for all its inputs every cycle, so the mechanism needs to be simple and fast.

Simple arithmetic operations are sufficient to select the output port in most regular topologies. For example, in a 2D mesh each packet will typically carry the signed distance to travel in each dimensions, $[\Delta x, \Delta y]$, in the packet header. The routing operation at switch ij is given by:

Direction	Condition
West $(-x)$	$\Delta x < 0$
East $(+x)$	$\Delta x > 0$
South $(-y)$	$\Delta x = 0, \Delta y < 0$
North $(+y)$	$\Delta x = 0, \Delta y > 0$
Processor	$\Delta x = 0, \Delta y = 0$

To accomplish this kind of routing, the switch needs to test the address in the header and decrement or increment one routing field. For a binary cube, the switch computes the position of the first bit that differs between the destination and the local node address, or the first non-zero bit if the packet carries the relative address of the destination. This kind of mechanism is used in Intel and nCUBE hypercubes, the Paragon, the Cal Tech Torus Routing Chip[Sesu93], and the J-machine, among others.

A more general approach is *source based routing*, in which the source builds a header consisting of the output port number for each switch along the route, $p_0, p_1, ..., p_{h-1}$. Each switch simply strips off the port number from the front of the message and sends the message out on the specified channel. This allows a very simple switch with little control state and without even arithmetic units to support sophisticated routing functions on arbitrary topologies. All of the intelligence is in the host nodes. It has the disadvantage that the header tends to be large, and usually of variable size. If the switch degree is d and routes of length h are permitted, the header may need to carry $h\log d$ routing bits. This approach is used in MIT Parc and Arctic routers, Meiko CS-2, and Myrinet.

A third approach, which is general purpose and allows for a small fixed size header is *table driven routing*, in which each switch contains a routing table, R, and the packet header contains a routing field, i, so the output port is determined by indexing into the table by the routing field, $o = R[i]$. This is used, for example, in HPPI and ATM switches. Generally, the table entry also gives the routing field for the next step in the route, $o, i' = R[i]$, to allow more flexibility in the table configuration. The disadvantage of this approach is that the switch must contain a sizable amount of routing state, and it requires additional switch-specific messages or some other mechanism for establishing the contents of the routing table. Furthermore, fairly large tables are required to simulate even simple routing algorithms. This approach is better suited to LAN and WAN traffic, where only a few of the possible routes among the collection of nodes are used at a time and these are mostly long lasting connections. By contrast, in a parallel machine there is often traffic among all the nodes.

Traditional networking routers contain a full processor which can inspect the incoming message, perform an arbitrary calculation to select the output port, and build a packet containing the message data for that output. This kind of approach is often employed in routers and (sometimes) bridges, which connect completely different networks (e.g., those that route between ethernet, FDDI, ATM) or at least different data link layers. It really does not make sense at the time scale of the communication within a high performance parallel machine.

10.5.2 Deterministic Routing

A routing algorithm is *deterministic* (or *nonadaptive*) if the route taken by a message is determined solely by its source and destination, and not by other traffic in the network. For example, dimension order routing is deterministic; the packet will follow its path regardless of whether a link along the way is blocked. Adaptive routing algorithms allow the route for a packet to be influenced by traffic it meets along the way. For example, in a mesh the route could zig-zag towards its destination if links along the dimension order path was blocked or faulty. In a fat tree, the upward path toward the common ancestor can steer away from blocked links, rather than following a random path determined when the message was injected into the network. If a routing algorithm only selects shortest paths toward the destination, it is *minimal*; otherwise it is nonminimal. Allowing multiple routes between each source and destination is clearly required for fault tolerance, and it also provides a way to spread load over more links. These virtues are enjoyed by source-based and table driven routing, but the choice is made when the packet is injected. Adaptive routing delays the choice until the packet is actually moving through the switch, which clearly makes the switch more complex, but has the potential of obtaining better link utilization.

We will concentrate first on deterministic routing algorithms and develop an understanding of some of the most popular routing algorithms and the techniques for proving them deadlock free, before investigating adaptive routing.

10.5.3 Deadlock Freedom

In our discussions of network latency and bandwidth we have tacitly assumed that messages make forward progress so it is meaningful to talk about performance. In this section, we show how one goes about proving that a network is deadlock free. Recall, *deadlock* occurs when a packet waits for an event that cannot occur, for example when no message can advance toward its destination because the queues of the message system are full and each is waiting for another to make resources available. This can be distinguished from *indefinite postponement*, which occurs when a packet awaits for an event that can occur, but never does, and from livelock, which occurs when the routing of a packet never leads to its destination. Indefinite postponement is primarily a question of fairness and livelock can only occur with adaptive nonminimal routing. Being free from deadlock is a basic property of well-designed networks that must be addressed from the very beginning.

Deadlock can occur in a variety of situations. A 'head-on' deadlock may occur when two nodes attempt to send to each other and each begins sending before either receives. It is clear that if they both attempt to complete sending before receiving, neither will make forward progress. We saw this situation at the user message passing layer using synchronous send and receive, and we saw it at the node-to-network interface layer. Within the network it could potentially occur with half duplex channels, or if the switch controller were not able to transmit and receive simultaneously on a bidirectional channel. We should think of the channel as a shared resource, which is acquired incrementally, first at the sending end and then at the receiver. In each case, the solution is to ensure that nodes can continue to receive while being unable to send. One could make this example more elaborate by putting more switches and links between the two nodes, but it is clear that a reliable network can only be deadlock free if the nodes are able to remove packets from the network even when they are unable to send packets. (Alternatively, we might recover from the deadlock by eventually detecting a timeout and aborting one or more of the packets, effectively

preempting the claim on the shared resource. This does raise the possibility of indefinite post-ponement, which we will address later.) In this head-on situation there is no routing involved, instead the problem is due to constraints imposed by the switch design.

A more interesting case of deadlock occurs when there are multiple messages competing for resources within the network, as in the routing deadlock illustrated in Figure 10-19. Here we have several messages moving through the network, where each message consists of several phits. We should view each channel in the network as having associated with it a certain amount of buffer resources; these may be input buffers at the channel destination, output buffers at the channel source, or both. In our example, each message is attempting to turn to the left and all of the packet buffers associated with the four channels are full. No message will release a packet buffer until after it has acquired a new packet buffer into which it can move. One could make this example more elaborate by separating the switches involved by additional switches and channels, but it is clear that the channel resources are allocated within the network on a distributed basis as a result of messages being routed through. They are acquired incrementally and non-preemptible, without losing packets.

This routing deadlock can occur with store-and-forward routing or even more easily with cut-through routing because each packet stretches over several phit buffers. Only the header flits of a packet carry routing information, so once the head of a message is spooled forward on a channel, all of the remaining flits of the message must spool along on the same channel. Thus, a single packet may hold onto channel resources across several switches. While we might hope to inter-leave packets with store-and-forward routing, the phits of a packet must be kept together. The essential point in these examples is that there are resources logically associated with channels and that messages introduce dependences between these resources as they move through the net-work.

The basic technique for proving a network deadlock free is to articulate the dependencies that can arise between channels as a result of messages moving through the networks and to show that there are no cycles in the overall channel dependency graph; hence there is no traffic patterns that can lead to deadlock. The most common way of doing this is to number the channel resources such that all routes follow a monotonically increasing (or decreasing) sequence, hence, no dependency cycles can arise. For a butterfly this is trivial, because the network itself is acyclic. It is also simple for trees and fat-trees, as long as the upward and downward channels are independent. For networks with cycles in the channel graph, the situation is more interesting.

To illustrate the basic technique for showing a routing algorithm to be deadlock-free, let us show that $\Delta x, \Delta y$ routing on a k-ary 2D array is deadlock free. To prove this, view each bidirectional channel as a pair of unidirectional channels, numbered independently. Assign each positive-x channel $\langle i, y \rangle \rightarrow \langle i+1, y \rangle$ the number i, and similarly number the negative-x going channel starting from 0 at the most positive edge. Number the positive-y channel $\langle x, j \rangle \rightarrow \langle x, j+1 \rangle$ the number $N + j$, and similarly number the negative-y edges from the from the most positive edge. This numbering is illustrated in Figure 10-20. Any route consisting of a sequence of consecutive edges in one x direction, a 90 degree turn, and a sequence of consecutive edges in one y direction is strictly increasing. The channel dependence graph has a node for every unidirectional link in the network and there is an edge from node A to node B if it is possible for a packet to traverse channel A and then channel B. All edges in the channel dependence graph go from lower numbered nodes to higher numbered ones, so there are no cycles in the channel dependence graph.
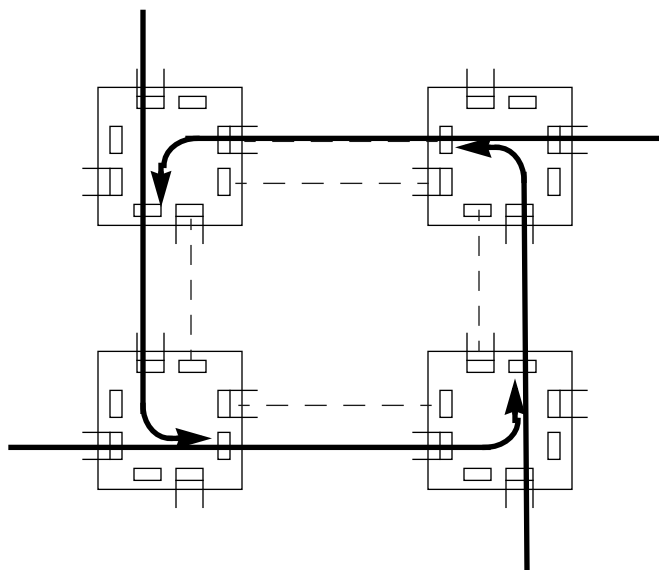
Figure 10-19 Examples of Network Routing Deadlock

Each of four switches has four input and output ports. Four packets have each aquired an input port and output buffer and an input port, and all are attempting to acquire the next output buffer as they turn left. None will relenquish their output buffer until it moves forward,so none can progress.

This proof easily generalizes to any number of dimensions and, since a binary d-cube consists of a pair of unidirectional channels in each dimension, also shows that e-cube routing is deadlock free on a hypercube. Observe, however, that the proof does not apply to k-ary d-cubes in general, because the channel number decreases at the wrap-around edges. Indeed, it is not hard to show that for $k > 4$, dimension order routing will introduce a dependence cycle on a unidirectional torus, even with $d = 1$.

Notice that deadlock-free routing proof applies even if there is only a single flit of buffering on each channel and that the potential for deadlock exists in k-ary d-cube even with multiple packet buffers and store-and-forward routing, since a single message may fill up all the packet buffers along its route. However, if the use of channel resources is restricted, it is possible to break the deadlock. For example, consider the case of a unidirectional torus with multiple packet buffers per channel and store-and-forward routing. Suppose that one of the packet buffers associated with each channel is reserved for messages destined for nodes with a larger number than their source, *i.e.*, packets that do not use wrap-around channels. This means that it will always be possible for "forward going" messages to make progress. Although wrap-around messages may be postponed, the network does not deadlock. This solution is typical of the family of techniques for making store-and-forward packet-switched networks deadlock free; there is a concept of a *structured buffer pool* and the routing algorithm restricts the assignment of buffers to packets to break dependence cycles. This solution is not sufficient for worm-hole routing, since it tacitly assumes that packets of different messages can be interleaved as they move forward.

Observe that deadlock free routing does not mean the system is deadlock free. The network is deadlock free only as long as it is drained into the NIs, even when the NIs are unable to send. If

Channel Dependence Graph

Unidirectional Version of 4-ary 2-array
with channels numbered
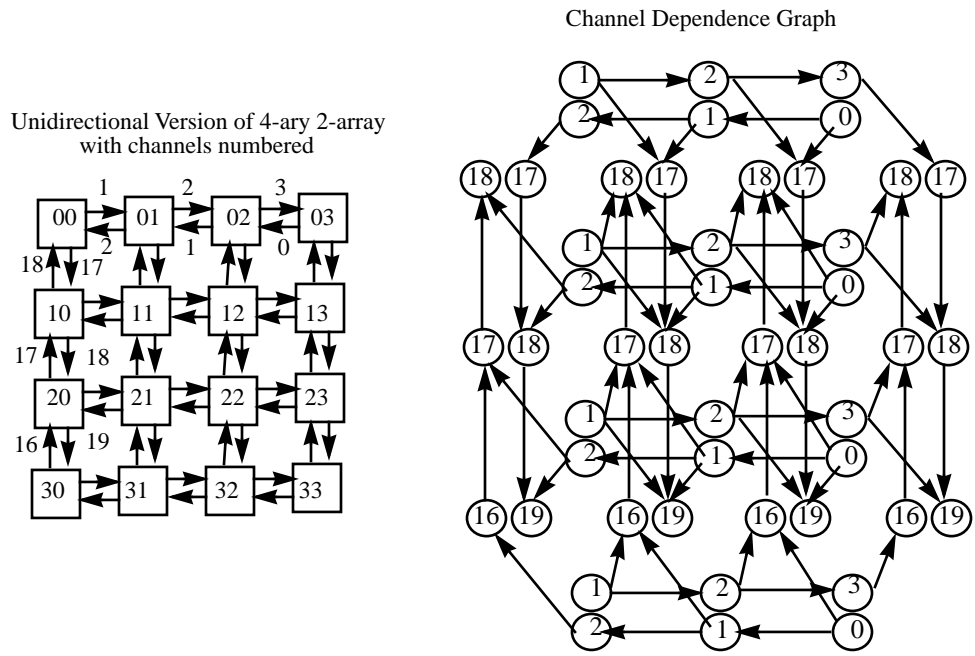
Figure 10-20 Channel ordering in the network graph and corresponding channel dependence graph

To show the routing algoirithm is deadlock free, it is sufficient to demonstrate that the channel dependence graph has no cycles.

two-phase protocols are employed, we need to ensure that fetch-deadlock is avoided. This means either providing two logically independent networks, or ensuring that the two phases are decoupled through NI buffers. Of course, the program may still have deadlocks, such as circular-waits on locks or head-on collision using synchronous message passing. We have worked our way down from the top, showing how to make each of these layers deadlock free, as long as the next layer is deadlock free.

Given a network topology and a set of resources per channel, there are two basic approaches for constructing a deadlock-free routing algorithm: restrict the paths that packets may follow or restrict how resources are allocated. This observation raises a number of interesting questions. Is there a general technique for producing deadlock free routes with wormhole routing on an arbitrary topology? Can such routes be adaptive? Is there some minimum amount of channel resources required?

10.5.4 Virtual Channels

The basic technique making networks with worm-hole routing deadlock free is to provide multiple buffers with each physical channel and to split these buffers into a group of *virtual channels*. Going back to our basic cost model for networks, this does not increase the number of links in the network, nor the number of switches. In fact, it does not even increase the size of the cross-bar internal to each switch, since only one flit at a time moves through the switch for each output channel. As indicated in Figure 10-21, it does require additional selectors and multiplexors

within the switch to allow the links and the cross-bar to be shared among multiple virtual channels per physical channel.

Figure 10-21 Multiple Virtual Channels in a basic switch
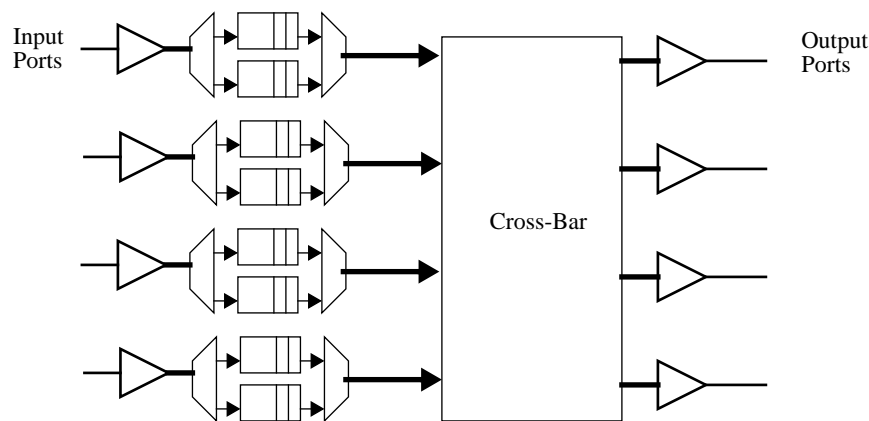
Each physical channel is shared by multiple virtual channels. The input ports of the switch split the incoming virtual channels into separate buffers, however, these are multiplexed through the switch to avoid expanding the cross-bar.

Virtual channels are used to avoid deadlock by systematically breaking cycles in the channel dependency graph. Consider, for example, the four-way routing deadlock cycle of Figure 10-19. Suppose we have two virtual channels per physical and messages at a node numbered higher than their destination are routed on the high channels, while messages at a node numbered less than their destinations are routed on the low channels. As illustrated in Figure 10-22, the dependence cycle is broken. Applying this approach to the k-ary d-cube, treat the channel labeling as a radix $d + 1 + k$ number of the form ivx, where i is the dimension, x is the coordinate of the source node of the channel in dimension i, and v is the virtual channel number. In each dimension, if the destination node has a smaller coordinate that the source node in that dimension, i.e., the message must use a wrap-around edge, use the $v = 1$ virtual channel in that dimension. Otherwise, use the $v = 0$ channel. The reader may verify that dimension order routing is deadlock free with this assignment of virtual channels. Similar techniques can be employed with other popular topologies [DaSe87]. Notice that with virtual channels we need to view the routing algorithm as a function $R : C \times N \to C$, because the virtual channel selected for the output depends on which channel it came in on.

10.5.5 Up*-Down* Routing

Are virtual channels required for deadlock free wormhole routing on an arbitrary topology? No. If we assume that all the channels are bidirectional there is a simple algorithm for deriving deadlock free routes for an arbitrary topology. Not surprisingly, it restricts the set of legal routes. The general strategy is similar to routing in a tree, where routes go up the tree away from the source and then down to the destination. We assume the network consists of a collection of switches, some of which have one or more hosts attached to them. Given the network graph, we want to number the switches so that the numbers increase as we get farther away from the hosts. One
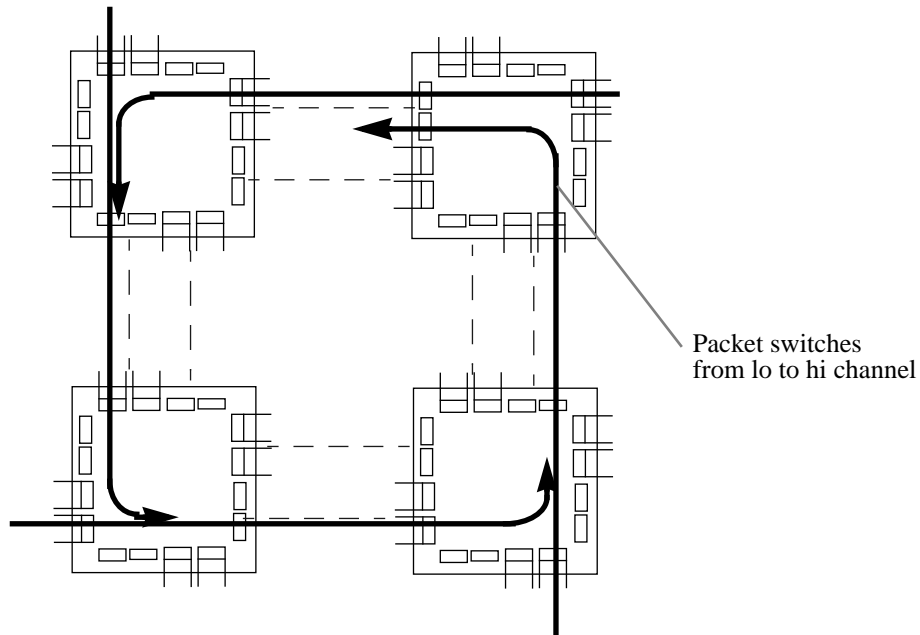
Figure 10-22 Breaking Deadlock Cycles with Virtual Channels

Each physical channel is broken into two virtual channels, call them lo and hi. The virtual channel "parity" of the input port is used for the output, except on turns north to west, which make a transition from lo to hi.

approach is to partition the network into levels, with the host nodes at level zero and each switch at the level corresponding to its maximum distance from a host. Assign numbers starting with level 0 and increasing with level. This numbering is easily accomplished with a breadth-first-search of the graph. (Alternatively, construct a spanning tree of the graph with hosts at the leaves and numbers increasing toward the root.) It is clear that for any source host, any destination host can be reached by a path consisting of a sequence of up channels (toward higher numbered nodes), a single turn, and a series of down channels. Moreover, the set of routes following such up*-down* paths are deadlock free. The network graph may have cycles, but the channel dependence graph under up*-down* routing does not. The up channels form a DAG and the down channels form a DAG. The up channels depend only on lower numbered up channels and the down channels depend only on up channels and higher numbered down channels.

This style of routing was developed for Autonet[And*92], which was intended to be self config-uring. Each of the switches contained a processor, which could run a distributed algorithm to determine the topology of the network and find a unique spanning tree. Each of the hosts would compute up*-down* routes as a restricted shortest-paths problem. Then routing tables in the switches could be established. The breadth-first search version is used by Atomic[Fel94] and Myrinet[Bod*95], where the switches are passive and the hosts determine the topology by prob-ing the network. Each host runs the BFS and shortest paths algorithms to determine its set of source-based routes to the other nodes. A key challenge in automatic mapping of networks, espe-cially with simple switches that only move messages through without any special processing is determining when two distinct routes through the network lead to the same switch[Mai*97]. One solution is to try returning to the source using routes from previously known switches, another is

detecting when there exists identical paths from two, supposedly distinct switches to the same host.

10.5.6 Turn-model Routing

We have seen that a deadlock free routing algorithm can be constructed by restricting the set of routes within a network or by providing buffering with each channel that is used in a structured fashion. How much do we need to restrict the routes? Is there a minimal set of restrictions or a minimal combination of routing restrictions and buffers? An important development in this direction, which has not yet appeared in commercial parallel machines, is turn-model routing. Consider, for example, a 2D array. There are eight possible turns, which form two simple cycles, as shown in Figure 10-23. (The figure is illustrating cycles appearing in the network involving multiple messages. The reader should see that there is a corresponding cycle in the channel dependency graph.) Dimension order routing prevents the use of four of the eight turns – when traveling in $\mp x$ it is legal to turn in $\mp y$, but once a packet is traveling in $\mp y$ it can make no further turns. The illegal turns are indicated by grey lines in the figure. Intuitively, it should be possible to prevent cycles by eliminating only one turn in each cycle.
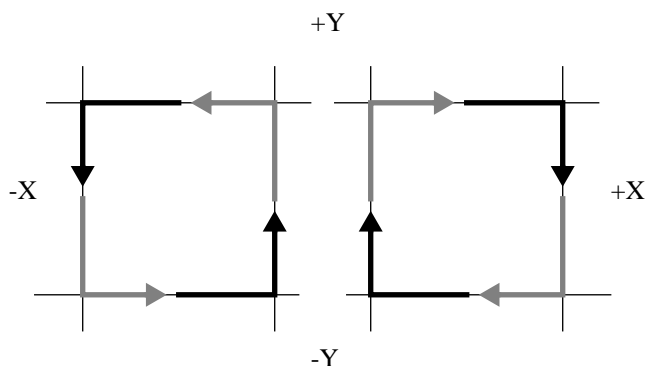
Figure 10-23 Turn Restrictions of $\Delta x, \Delta y$ routing

.Dimension order routing on a 2D array prohibits the use of four of the eight possible turns, thereby breaking both of the simple dependence cycles. A deadlock free routing algorithm can be obtained by prohibiting only two turns.

Of the 16 different ways to prohibit two turns in a 2D array, 12 prevent deadlock. These consist of the three unique shown in Figure 10-24 and rotations of these. The west-first algorithm is so named because there is no turn allowed into the $-x$ direction; therefore, if a packet needs to travel in this direction it must do so before making any turns. Similarly, in north-last there is no way to turn out of the $+y$ direction, so the route must make all its other adjustments before heading in this direction. Finally, negative-first prohibits turns from a positive direction into a negative direction, so the route must go as negative as its needs to before heading in either positive direction.

Each of these turn-model algorithms allow very complex, even non-minimal routes. For example, Figure 10-23 shows some of the routes that might be taken under west-first routing. The elongated rectangles indicate blockages or broken links that might cause such a set of routes to be
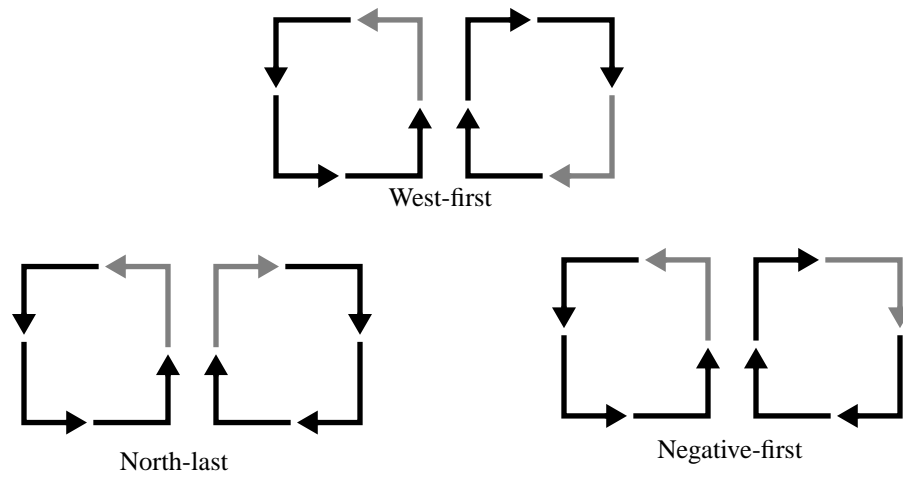
Figure 10-24 Minimal Turn Model Routing in 2D

Only two of the eight possible turns need be prohibited in order to obtain a deadlock free routing algorithm. Legal turns are shown for three such algorithms.

used. It should be clear that minimal turn models allow a great deal of flexibility in route selection. There are many legal paths between pairs of nodes.
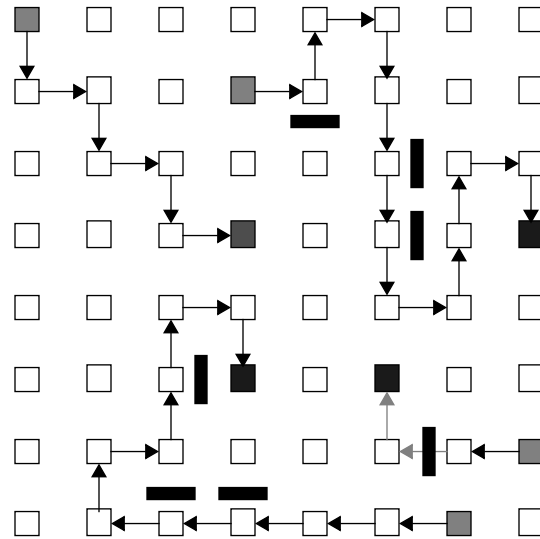
Figure 10-25 Examples of legal west-first routes in a 8x8 array

Substantial routing adaptation is obtained with turn model routing, thus providing the ability to route around faults in a deadlock free manner.

The turn-model approach can be combined with virtual channels and it can be applied in any topology. (In some networks, such as unidirectional d-cubes, virtual channels are still required.) The basic method is (1) partition the channels into sets according to the direction they route packets (excluding wraparound edges), (2) identify the potential cycles formed by "turns" between directions, and (3) prohibit one turn in each abstract cycle, being careful to break all the complex cycles as well. Finally, wraparound edges can be incorporated, as long as they do not introduce cycles. If virtual channels are present, treat each set of channels as a distinct virtual direction.

Up*-down* is essentially a turn-model algorithm, with the assumption of bidirectional channels, using only two directions. Indeed, reviewing the up*-down* algorithm, there may be many shortest paths that conform to the up*-down* restriction, and there are certainly many non-minimal up*-down* routes in most networks. Virtual channels allow the routing restrictions to be loosened even further.

10.5.7 Adaptive Routing

The fundamental advantage of loosening the routing restrictions is that it allows there to be multiple legal paths between pairs of nodes. This is essential for fault tolerance. If the routing algorithm allows only one path, failure of a single link will effectively leave the network disconnected. With multipath routing, it may be possible to steer around the fault. In addition, it allows traffic to be spread more broadly over available channels and thereby improves the utilization of the network. When a vehicle is parked in the middle of the street, it is often nice to have the option of driving around the block.

Simple deterministic routing algorithms can introduce tremendous contention within the network, even though communication load is spread evenly over independent destinations. For example, Figure 10-26 shows a simple case where four packets are traveling to distinct destinations in a 2D mesh, but under dimension order routing they are all forced to travel through the same link. The communication is completely serialized through the bottleneck, while links for other shortest paths are unused. A multipath routing algorithm could use alternative channels, as indicated in the right hand portion of the figure. For any network topology there exist bad permutations[GoKr84], but simple deterministic routing makes these bad permutations much easier to run across. The particular example in Figure 10-26 has important practical significance. A common global communication pattern is a transpose. On a 2D mesh with dimension order routing, all the packets in a row must go through a single switch before filling in the column.

Multipath routing can be incorporated as an extension of any of the basic switch mechanisms. With source based routing, the source simply chooses among the legal routes and builds the header accordingly. No change to the switch is required. With table driven routing, this can be accomplished by setting up table entries for multiple paths. For arithmetic routing additional control information would need to be incorporated in the header and interpreted by the switch.

Adaptive routing is a form of multipath routing where the choice of routes is made dynamically by the switch in response to traffic encountered en route. Formally, an adaptive routing function is a mapping of the form: $R_A : C \times N \times \Sigma \to C$, where Σ represents the switch state. In particular, if one of the desired outputs is blocked or failed, the switch may choose to send the packet on an alternative channel. Minimal adaptive routing will only route packets along shortest paths to their destination, i.e., every hop must reduce the distance to the destination. An adaptive algorithm that allows all shortest paths to be used is *fully adaptive*, otherwise it is *partially adaptive*. An inter-
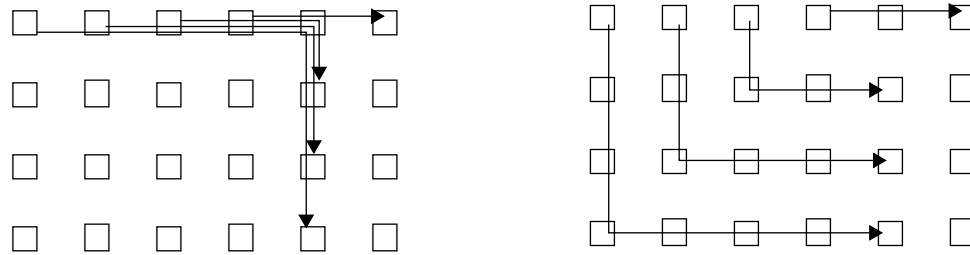
Figure 10-26 Routing path conflicts under deterministic dimension-order routing

Several message from distinct source to distinct destination contend for resources under dimension order routing, whereas an adaptive routing scheme may be able to use disjoint paths.

esting extreme case of non-minimal adaptive routing is what is called "hot potato" routing. In this scheme the switch never buffers packets. If more than one packet is destined for the same output channel, the switch sends one towards its destination and *misroutes* the rest onto other channels.

Adaptive routing is not widely used in current parallel machines, although it has been studied extensively in the literature[NgSe89,LiHa91], especially through the Chaos router[KoSn91]. The nCUBE/3 is to provide minimal adaptive routing in a hypercube. The network proposed for the Tera machine is to use hot potato routing, with 128 bit packets delivered in one 3ns cycle.

Although adaptive routing has clear advantages, it is not without its disadvantages. Clearly it adds to the complexity of the switch, which can only make the switch slower. The reduction in bandwidth can outweigh the gains of the more sophisticated routing – a simple deterministic net-work in its operating regime is likely to outperform a clever adaptive network in saturation. In non-uniform networks, such as a d-dimensional array, adaptivity hurts performance on uniform random traffic. Stochastic variations in the load introduce temporary blocking in any network. For switches at the boundary of the array, this will tend to propel packets toward the center. As a result, contention forms in the middle of the array that is not present under deterministic routing.

Adaptive routing can cause problems with certain kinds of non-uniform traffic as well. With adaptive routing, packets destined for a hot spot will bounce off the saturated path and clog up the rest of the network, not just the tree to the destination, until all traffic is effected. In addition, when the hot spot clears, it will take even longer for the network saturation to drain because more packets destined for the problem node are in the network.

Non-minimal adaptive routing performs poorly as the network reaches saturation, because pack-ets traverse extra links and hence consume more bandwidth. The throughput of the network tends to drop off as load is increased, rather than flattening at the saturation point, as suggested by Figure 10-17.

Recently there have been a number of proposals for low-cost partially and fully adaptive that use a combination of a limited number of virtual channels and restrictions on the set of turns[ChKi92, ScJa95]. It appears that most of the advantages of adaptive routing, including fault tolerance and channel utilization, can be obtained with a very limited degree of adaptability.

10.6 Switch Design

Ultimately, the design of a network boils down to the design of the switch and how the switches are wired together. The degree of the switch, its internal routing mechanisms, and its internal buffering determine what topologies can be supported and what routing algorithms can be implemented. Now that we understand these higher level network design issues, let us return to the switch design in more detail. Like any other hardware component of a computer system, a network switch comprises: datapath, control, and storage. This basic structure was illustrated at the beginning of the chapter in Figure 10-5. Throughout most of the history of parallel computing, switches were built from a large number of low integration components, occupying a board or a rack. Since the mid 1980s, most parallel computer networks are built around single chip VLSI switches – exactly the same technology as the microprocessor. (This transition is taking place in LANs a decade later.) Thus, switch design is tied to the same technological trends discussed in Chapter 1: decreasing feature size, increasing area, and increasing pin count. We should view modern chip design from a VLSI perspective.

10.6.1 Ports

The total number of pins is essentially the total number of input and output ports times the channel width. Since the perimeter of the chip grows slowly compared to area, switches tend to be pin limited. The pushes designers toward narrow high frequency channels. Very high speed serial links are especially attractive, because it uses the least pins and eliminates problems with skew across the bit lines in a channel. However, with serial links the clock and all control signals must be encoded within the framing of the serial bit stream. With parallel links, one of the wires is essentially a clock for the data on the others. Flow control is realized using an additional wire, providing a ready, acknowledge handshake.

10.6.2 Internal Datapath

The datapath is the connectivity between each of a set of input ports, *i.e.,* input latches, buffers or FIFO, and every output port. This is generally referred to as the internal cross-bar, although it can be realized in many different ways. A *non-blocking cross-bar* is one in which each input port can be connected to a distinct output in any permutation simultaneously. Logically, for an $n \times n$ switch the non-blocking cross-bar is nothing more than an n-way multiplexor associated with each destination, as shown in Figure 10-27. The multiplexor may be implemented in a variety of different ways, depending on the underlying technology. For example, in VLSI it is typically realized as a single bus with n tristate drivers, also shown in Figure 10-27. In this case, the control path provides n selects per output.

It is clear that the hardware complexity of the cross bar is determined by the wires. There are nw data wires in each direction, requiring $(nw)^2$ area. There are also n^2 control wires, which add to this significantly. How do we expect switches to track improvements in VLSI technology? Assume that the area of the cross-bar stays constant, but the feature size decreases. The ideal VLSI scaling law says that if the feature size is reduced by a factor of s (including the gate thickness) and reduce the voltage level by the same factor, then the speed of the transistors improves by a factor of $1/s$, the propagation delay of wires connecting neighboring transistors improves by a factor of $1/s$, and the total number of transistors per unit area increases by $1/s^2$ with the same power density. For switches, this means that the wires get thinner and closer together, so the
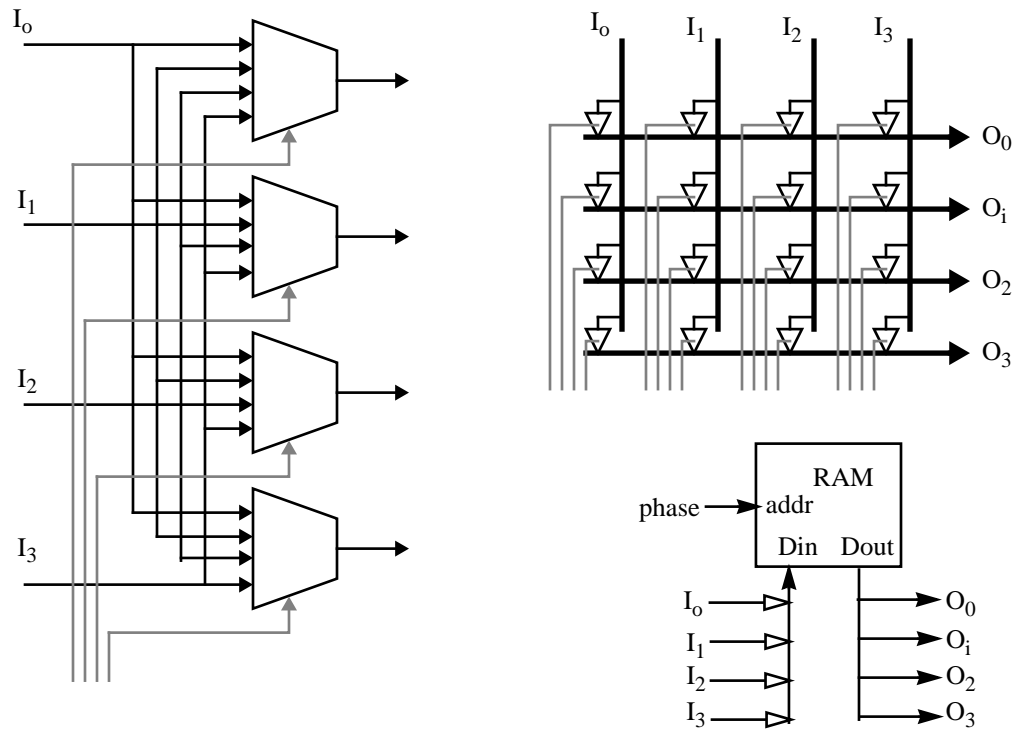
Figure 10-27 Cross-bar Implementations

The cross bar internal to a switch can be implemented as a collection of multiplexors, a grid of tristate drivers, or via a contentional static RAM that time multiplexes across the ports.

degree of the switch can increase by a factor of $1/s$. Notice, the switch degree improves as only the square root of the improvement in logic density. The bad news is that these wires run the entire length of the cross bar, hence the length of the wires stays constant. The wires get thinner, so they have more resistance. The capacitance is reduced and the net effect is that the propagation delay is unchanged[Bak90]. In other words, ideal scaling gives us an improvement is switch degree for the same area, but no improvement is speed. The speed does improve (by a factor of $1/s$ if the voltage level is held constant, but then the power increases with $1/s^2$. Increases in chip area will allow larger degree, but the wire lengths increase and the propagation delays will increase.

There is some degree of confusion over the term cross bar. In the traditional switching literature multistage interconnection network that have a single controller are sometimes called cross bars, eventhough the data path through the switch is organized as an interconnection of small cross bars. In many cases these are connected in a butterfly topology, called a Banyan network in the switching literature. A banyan networks is non-blocking if its inputs are sorted, so some non-blocking cross-bars are built as batcher sorting network in front of a banyan network. An approach that has many aspects in common with Benes networks is to employ a variant of the butterfly, called a delta network, and use two of these networks in series. The first serves to randomize packets position relative to the input ports and the second routes to the output port. This is used, for example, in some commercial ATM switches[Tu888]. In VLSI switches, it is usually

more effective to actually build a non-blocking cross-bar, since its is simple, fast and regular. The key limit is pins anyways.

It is clear that VLSI switches will continue to advance with the underlying technology, although the growth rate is likely to be slower than the rate of improvement in storage and logic. The hardware complexity of the cross-bar can be reduced if we give up the non-blocking property and limit the number of inputs that can be connected to outputs at once. In the extreme, this reduces to a bus with n drivers and n output selects. However, the most serious issue in practice turns out to be the length of the wires and the number of pins, so reducing the internal bandwidth of the individual switches provides little savings and a significant loss in network performance.

10.6.3 Channel Buffers

The organization of the buffer storage within the switch has a significant impact on the switch performance. Traditional routers and switches tend to have large SRAM or DRAM buffers external to the switch fabric, whereas in VLSI switches the buffering is internal to the switch and comes out of the same silicon budget as the datapath and the control section. There are four basic options: no buffering (just input and output latches), buffers on the inputs, buffers on the outputs, or a centralized shared buffer pool. A few flits of buffering on input and output channels decouples the switches on either end of the link and tends to provide a significant improvement in performance. As chip size and density increase, more buffering is available and the network designer has more options, but still the buffer real-estate comes at a premium and its organization is important. Like so many other aspects in network design, the issue is not just how effectively the buffer resources are utilized, but how the buffering effects the utilization of other components of the network.

Intuitively, one might expect sharing of the switch storage resources to be harder to implement, but allow better utilization of these resources than partitioning the storage among ports. They are indeed harder to implement, because all of the communication ports need to access the pool simultaneously, requiring a very high bandwidth memory. More surprisingly, sharing the buffer pool on demand can hurt the network utilization, because a single congested output port can "hog" most of the buffer pool and thereby prevent other traffic from moving through the switch. Given the trends in VLSI, it makes sense to keep the design simple and throw a little more independent resources at the problem.

Input buffering

An attractive approach is to provide independent FIFO buffers with each input port, as illustrated in Figure 10-28. Each buffer needs to be able to accept a flit every cycle and deliver one flit to an output, so the internal bandwidth of the switch is easily matched to the flow of data coming in. The operation of the switch is relatively simple; it monitors the head of each input fifo, computes the desired output port of each, and schedules packets to move through the cross-bar accordingly. Typically, there is routing logic associated with each input port to determine the desired output. This is trivial for source-based routing; it requires an arithmetic unit per input for algorithmic routing, and, typically a routing table per input with table-driven routing. With cut-through routing, the decision logic does not make an independent choice every cycle, but only every packet. Thus, the routing logic is essentially a finite state machine, which spools all the flits of a packet to the same output channel before making a new routing decision at the packet boundary[SeSu93].
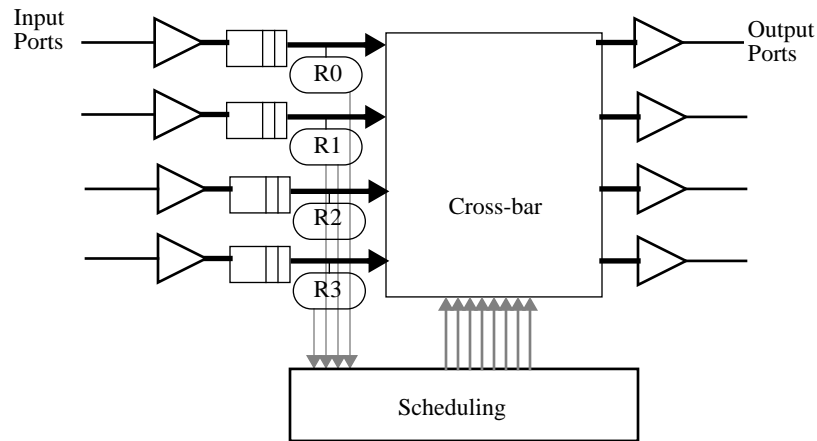
Figure 10-28 Input Buffered Switch

A FIFO is provided at each of the input ports, but the controller can only inspect and service the packets at the heads of the input FIFOs.

One problem with the simple input buffered approach is the occurrence of "head of line" (HOL) blocking. Suppose that two ports have packets destined for the same output port. One of them will be scheduled onto the output and the other will be blocked. The packet just behind the blocked packet may be destined for one of the unused outputs (there is guaranteed to be unused outputs), but it will not be able to move forward. This head-of-line blocking problem is familiar in our vehicular traffic analogy, it corresponds to having only one lane approaching an intersection. If the car ahead is blocked attempting to turn there is no way to proceed down the empty street ahead.

We can easily estimate the effect of head-of-line blocking on channel utilization. If we have two input ports and randomly pick an output for each, the first succeeds and the second has a 50-50 chance of picking the unused output. Thus, the expected number of packets per cycle moving through the switch is 1.5 and, hence, the expected utilization of each output is 75%. Generalizing this, if $E(n, k)$ is the expect number of output ports covered by k random inputs to an n-port switch then, $E(n, k + 1) = E(n, k) + \frac{n - E(n, k)}{n}$. Computing this recurrence up to $k = n$ for various switch sizes reveals that the expected output channel utilization for a single cycle of a fully loaded switch quickly drops to about 65%. Queueing theory analysis shows that the expected utilization in steady state with input queueing is 59%[Kar*87].

The impact of HOL blocking can be more significant than this simple probabilistic analysis indicates. Within a switch, there may be bursts of traffic for one output followed by bursts for another, and so on. Even though the traffic is evenly distributed, given a large enough window, each burst results in blocking on all inputs.[Li88] Even if there is no contention for an output within a switch, the packet at the head of an input buffer may be destined for an output that is blocked due to congestion elsewhere in the network. Still the packets behind it cannot move forward. In a wormhole routed network, the entire worm will be stuck in place, effectively consum-

ing link bandwidth without going anywhere. A more flexible organization of buffer resources might allow packets to slide ahead of packets that are blocked.

Output buffering

The basic enhancement we need to make to the switch is to provide a way for it to consider multiple packets at each input as candidates for advancement to the output port. A natural option is to expand the input FIFOs to provide an independent buffer for each output port, so packets sort themselves by destination upon arrival, as indicated by Figure 10-29. (This is the kind of switch assumed by the conventional delay analysis in Section 10.4; the analysis is simplified because the switch does not introduce additional contention effects internally.) With a steady stream of traffic on the inputs, the outputs can be driven at essentially 100%. However, the advantages of such a design are not without a cost; additional buffer storage and internal interconnect is required.[1] Along with the sorting stage and the wider multiplexors, this may increase the switch cycle time or increase its routing delay.

It is a matter of perspective whether the buffers in Figure 10-29 are associated with the input or the output ports. If viewed as output port buffers, the key property is that each output port has enough internal bandwidth to receive a packet from every input port in one cycle. This could be obtained with a single output FIFO, but it would have to run at an internal clock rate of n times that of the input ports.

Virtual Channels buffering

Virtual channels suggest an alternative way of organizing the internal buffers of the switch. Recall, a set of virtual channels provide transmission of multiple independent packets across a single physical link. As illustrated in Figure 10-29, to support virtual channels the flow across the link is split upon arrival at an input port into distinct channel buffers. (These are multiplexed together again, either before or after the crossbar, onto the output ports. If one of the virtual channel buffers is blocked, it is natural to consider advancing the other virtual channels toward the outputs. Again, the switch has the opportunity to select among multiple packets at each input for advance to the output ports, however, in this case the choice is among different virtual channels rather than different output ports. It is possible that all the virtual channels will need to route to the same output port, but expected coverage of the outputs is much better. In a probabilistic analysis, we can ask what the expected number of distinct output ports are covered by choosing among vn requests for n ports, where v is the number of virtual channels..

Simulation studies show that large (256 to 1024 node) 2-ary butterflies using wormhole routing with moderate buffering (16 flits per channel) saturate at a channel utilization of about 25% under random traffic. If the same sixteen flits of buffering per channel are distributed over a larger number of virtual channels, the saturation bandwidth increases substantially. It exceeds 40% with just two virtual channels (8-flit buffers) and is nearly 80% at sixteen channels with single-flit buffers[Dal90a]. While this study keeps the total buffering per channel fixed, it does not really

---

1. The MIT Parc switch[Joe94] provides the capability of the nxn buffered switch but avoid the storage and interconnect penalty. It is designed for fixed-size packets. The set of buffers at each input form a pool and each output has a list of pointers to packets destined for it. The timing constraint in this design is the ability to push n pointers per cycle into the output port buffer, rather than n packets per cycle.
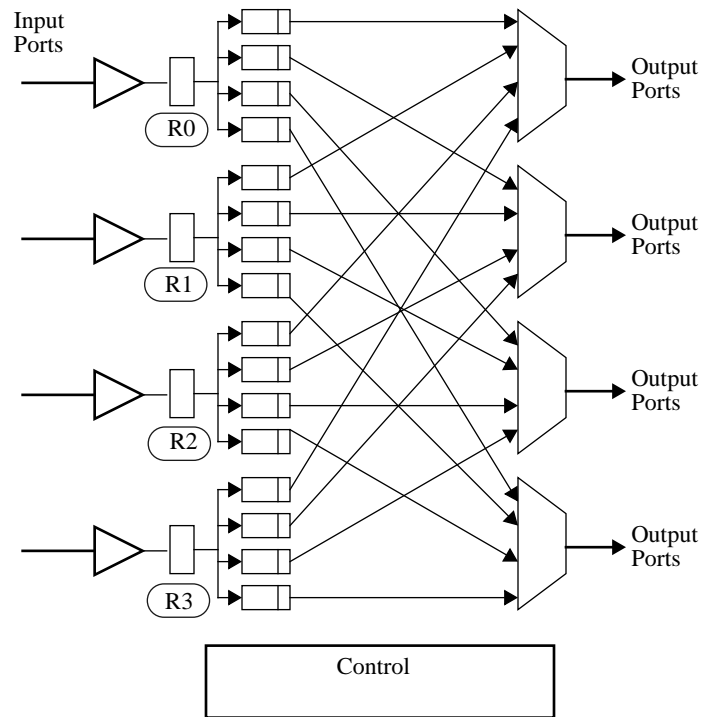
Figure 10-29 Switch Design to avoid HOL blocking

Packets are sorted by output port at each input port so that the controller can a packet for an output port if any input has a packet destined for that output.

keep the cost constant. Notice that a routing decision needs to be computed for each virtual channel, rather than each physical channel, if packets from any of the channels are to be considered for advancement to output ports.

By now the reader is probably jumping a step ahead to additional cost/performance trade-offs that might be considered. For example, if the cross-bar in Figure 10-27 has an input per virtual channel, then multiple packets can advance from a single input at once. This increases the probability of each packet advancing and hence the channel utilization. The cross-bar increases in size in only one dimension and the multiplexors are eliminated, since each output port is logically a *vn* -way mux.

10.6.4 Output Scheduling

We have seen routing mechanisms that determine the desired output port for each input packet, datapaths that provide a connection from the input ports to the outputs, and buffering strategies that allow multiple packets per input port to be considered as candidates for advancement to the output port. A key missing component in the switch design is the *scheduling algorithm* which selects which of the candidate packets advance in each cycle. Given a selection, the remainder of the switch control asserts the control points in the cross-bar/mux's and buffers/latches to effect

the register transfer from each selected inputs to the associated output. As with the other aspects of switch design, there is a spectrum of solutions varying from simple to complex.

A simple approach is to view the scheduling problem as n independent arbitration problems, one for each output port. Each candidate input buffer has a request line to each output port and a grant line from each port, as indicated by Figure 10-30. (The figure shows four candidate input buffers driving three output ports to indicate that routing logic and arbitration input is on a per input buffer basis, rather than per input port.) The routing logic computes the desired output port and asserts the request line for the selected output port. The output port scheduling logic arbitrates among the requests, selects one and asserts the corresponding grant signal. Specifically, with the cross-bar using tri-state drivers in Figure 10-27b output port j enables input buffer i by asserting control enable e_{ij}. The input buffer logic advances its FIFO as a result of one of the grant lines being asserted.
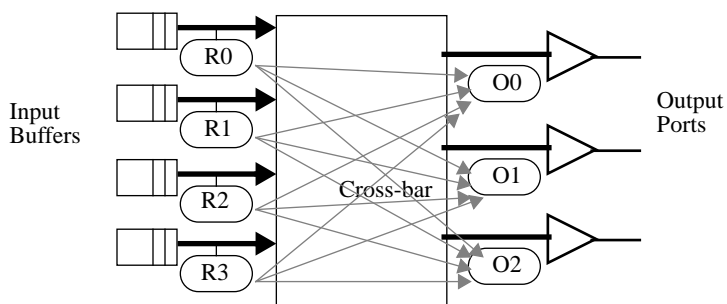
Figure 10-30 Control Structure for Output Scheduling

.Associated with each input buffer is routing logic to determine the output port, a request line and a grant line per output. Each output has selection logic to arbitrate among asserted request and assert one grant, causing a flit to advance from the input buffer to the output port.

An additional design question in the switch is the arbitration algorithm used for scheduling flits onto the output. Options include a static priority scheme, random, round-robin, and oldest-first (or *deadline* scheduling). Each of these have different performance characteristics and implementation complexity. Clearly static priority is the simplest; it is simply a priority encoder. However, in a large network it can cause indefinite postponement. In general, scheduling algorithms that provide fair service to the inputs perform better. Round-robin requires extra state to change the order of priority in each cycle. Oldest-first tends to have the same average latency as random assignment, but significantly reduces the variance in latencies[Dal90a]. One way to implement oldest-first scheduling, is to use a control FIFO of input port numbers at each output port. When an input buffer requests an output, the request is enqueued. The oldest request at the head of the FIFO is granted.

It is useful to consider the implementation of various routing algorithms and topologies in terms of Figure 10-30. For example, in a direct d-cube, there are $d+1$ inputs, let's number them $i_o, ..., i_d$, and $d+1$ outputs, numbered $o_1, ..., o_{d+1}$, with the host connected to input i_0 and output o_{d+1}. The straight path $i_j \rightarrow o_j$ corresponds to routing in the same dimension, other paths are a change in dimension. With dimension order routing, packets can only increase in dimension as they cross the switch. Thus, the full complement of request/grant logic is not required. Input j

need only request outputs $j, ..., d+1$ and output j need only grant inputs $0, ..., j$. Obvious static priority schemes assign priority in increasing or decreasing numerical order. If we further specialize the switch design to dimension order routing and use a stacked 2x2 cross-bar design (See exercise) then each input requests only two outputs and each output grants only two inputs. The scheduling issue amounts to preferring traffic continuing in a dimension vs. traffic turning onto the next higher dimension.

What are implementation requirements of adaptive routing? First, the routing logic for an input must compute multiple candidate outputs according to the specific rules of the algorithm, e.g., turn restrictions or plane restrictions. For partially adaptive routing, this may be only a couple of candidates. Each output receives several request and can grant one. (Or if the output is blocked, it may not grant any.) The tricky point is that an input may be selected by more than one output. It will need to choose one, but what happens to the other output? Should it iterate on its arbitration and select another input or go idle? This problem can be formalized as one of on-line bipartite matching[Kar*90]. The requests define a bipartite graph with inputs on one side and outputs on the other. The grants (one per output) define a matching of input-output pairs within the request graph. The maximum matching would allow the largest number of inputs to advance in a cycle, which ought to give the highest channel utilization. Viewing the problem in this way, one can construct logic that approximates fast parallel matching algorithms[And*92]. The basic idea is to form a tentative matching using a simple greedy algorithm, such as random selection among requests at each output followed by selection of grants at each inputs. Then, for each unselected output, try to make an improvement to the tentative matching. In practice, the improvement diminishes after a couple of iterations. Clearly, this is another case of sophistication vs. speed. It the scheduling algorithm increases the switch cycle time or routing delay, it may be better to accept a little extra blocking and get the job done faster.

This maximum matching problem applies to the case where multiple virtual channels are multiplexed through each input of the cross-bar, even with deterministic routing. (Indeed, the technique was proposed for the AN2 ATM switch to address the situation of scheduling cells from several "virtual circuits".[1] Each input port has multiple buffers that it can schedule onto its cross-bar input, and these may be destined for different outputs. The selection of the outputs determine which virtual channel to advance. If the cross-bar is widened, rather than multiplexing the inputs, the matching problem vanishes and each output can make a simple independent arbitration.

10.6.5 Stacked Dimension Switches

Many aspects of switch design are simplified if there are only two input and two outputs, including the control, arbitration, and data paths. Several designs, including the torus routing chip[SeSu93], J-machine, and Cray T3D, have used a simple 2x2 building block and stacked these to construct switches of higher dimension, as illustrated in Figure 10-31. If we have in mind a d-cube, traffic continuing in a given dimension passes straight through the switch in that dimension, whereas if it need to turn into another dimension it is routed vertically through the switch. Notice that this adds a hop for all but the lowest dimension. This same technique yields a topology called cube-connected cycles when applied to a hypercube.

_____

1. Virtual circuits should not be confused with virtual channels. The former is a technique for associating routing a resources along an entire source to destination route. The latter is a strategy for structuring the buffering associated with each link.
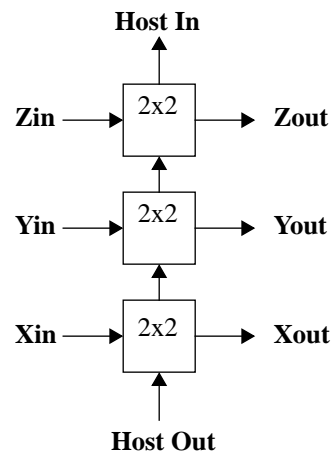
Figure 10-31 Dimension stacked switch

Traffic continuing in a dimension passes straight through one 2x2 switch, whereas when it turns to route in another dimension it is routed vertically up the stack

10.7 Flow Control

In this section we consider in detail what happens when multiple flows of data in the network attempt to use the same shared network resources at the same time. Some action must be taken to control these flows. If no data is to be lost, some of the flows must be blocked while others proceed. The problem of flow control arises in all networks and at many levels, but it is qualitatively different in parallel computer networks than in local and wide area networks. In parallel computers network traffic needs to be delivered about as reliably as traffic across a bus and there are a very large number of concurrent flows on very small time scales. No other networking regime has such stringent demands. We will look briefly at some of these differences and then examine in detail how flow control is addressed at the link level and end-to-end in parallel machines.

10.7.1 Parallel Computer Networks vs. LANs and WANs

To build intuition for the unique flow-control requirements of the parallel machine networks, let us take a little digression and examine the role of flow control in the networks we deal with every day for file transfer and the like. We will look at three examples: ethernet style collision based arbitration, FDDI style global arbitration, and unarbitrated wide-area networks.

In an ethernet, the entire network is effectively a single shared wire, like a bus only longer. (The aggregate bandwidth is equal to the link bandwidth.) However, unlike a bus there is no explicit arbitration unit. A host attempts to send a packet by first checking that the network appears to be quiet and then (optimistically) driving its packet onto the wire. All nodes watch the wire, including the hosts attempting to send. If there is only one packet one the wire, every host will "see" it and the host specified as the destination will pick it up. If there is a collision, every host will

detect the garbled signal, including the multiple senders. The minimum that a host may drive a packet, i.e., the minimum channel time, is about 50 μs; this is to allow time for all hosts to detect collisions.

The flow-control aspect is how the retry is handled. Each sender backs off for a random amount of time and then retries. With each repeated collision the retry interval from which the random delay is chosen is increased. The collision detection is performed within the network interface hardware, and the retry is handled by the lowest level of the ethernet driver. If there is no success after a large number of tries, the ethernet driver gives up and drops the packet. However, the message operation originated from some higher-level communication software layer which has its own delivery contract. For example, the TCP/IP layer will detect the delivery failure via a timeout and engage its own adaptive retry mechanism, just as it does for wide-area connection, which we discuss below. The UDP layer will ignore the delivery failure, leaving it to the user application to detect the event and retry. The basic concept of the ethernet rests on the assumption that the wire is very fast compared to the communication capability of the hosts. (This assumption was reasonable in the mid 70s when ethernet was developed.) A great deal of study has been given to the properties of collision-based media access control, but basically as the network reaches saturation, the delivered bandwidth drops precipitously.

Ring-based LANS, such as token-ring and FDDI, use a distributed form of global arbitration to control access to the shared medium. A special arbitration token circulates the ring when there is an empty slot. A host that desires to send a packet waits until the token comes around, grabs it, and drives the packet onto the ring. After the packet is sent, the arbitration token is returned to the ring. In effect, flow-control is performed at the hosts on every packet as part of gaining access to the ring. Even if the ring is idle, a host must wait for the token to traverse half the ring, on average. (This is why the unloaded latency for FDDI is generally higher than that of Ethernet.) However, under high load the full link capacity can be used. Again, the basic assumption underlying this global arbitration scheme is that the network operates on a much small timescale than the communication operations of the hosts.

In the wide-area case, each TCP connection (and each UDP packet) follows a path through a series of switches, bridges and routers across media of varying speeds between source and destination. Since the internet is a graph, rather than a simple linear structure, at any point there may be a set of incoming flows with a collective bandwidth greater than the outgoing link. Traffic will back up as a result. Wide-area routers provide a substantial amount of buffering to absorb stochastic variations in the flows, but if the contention persists these buffers will eventually fill up. In this case, most routers will just drop the packets. Wide-area links may be stretches of fibers many miles long, so when the packet is driven onto a link it is hard to know if it will find buffer space available when it arrives. Furthermore, the flows of data through a switch are usually unrelated transfers. They are not, in general, a collection of flows from within a single parallel program, which imposes a degree of inherent end-to-end flow control by waiting for data it needs before continuing. The TCP layer provides end-to-end flow control and adapts dynamically to the perceived characteristics of the route occupied by a connection. It assumes that packet loss (detected via timeout) is a result of contention at some intermediate point, so when it experiences a loss it sharply decreases the send rate (by reducing the size of its burst window). It slowly increases this rate, i.e., the window size, as data is transferred successfully (detected via acknowledgments from destination to source), until it once again experiences a loss. Thus, each flow is controlled at the source governed by the occurrence of time-outs and acknowledgments.

Of course, the wide-area case operates on timescale of fractions of a second, whereas parallel machine networks operate on a scale nanoseconds. Thus, one should not expect the techniques to carry over directly. Interestingly, the TCP flow-control mechanisms do work well in the context of collision-based arbitration, such as ethernet. (Partly the software overhead time tends to give a chance for the network to clear.) However, with the emergence of high-speed, switched local and wide area networks, especially ATM, the flow-control problem has taken on much more of the characteristics of the parallel machine case. At the time of writing, most commercial ATM switches provide a sizable amount of buffering per link (typically 64 to 128 cells per link) but drop cells when this is exceeded. Each cell is 53 bytes, so at the 155 Mb/s OC-3 rate a cell transfer time is 2.7 µs. Buffers fill very rapidly compared to typical LAN/WAN end-to-end times. (The TCP mechanisms prove to be quite ineffective in the ATM settings when contending with more aggressive protocols, such as UDP.) Currently, the ATM standardization efforts are hotly debating link-level flow and rate control measures. This development can be expected to draw heavily on the experiences from parallel machines.

10.7.2   Link-level flow control

Essentially all parallel machine interconnection networks provide link-level flow control. The basic problem is illustrated in Figure 10-32. Data is to be transferred from an output port of one node across a link to an input port of a node operating autonomously. The storage may be a simple latch, a FIFO, or buffer memory. The link may be short or long, wide or thin, synchronous or asynchronous. The key point is that as a result of circumstances at the destination node, storage at the input port may not be available to accept the transfer, so the data must be retained at the source until the destination is ready. This may cause the buffers at the source to fill, and it in turn may exert "back pressure" on its sources.
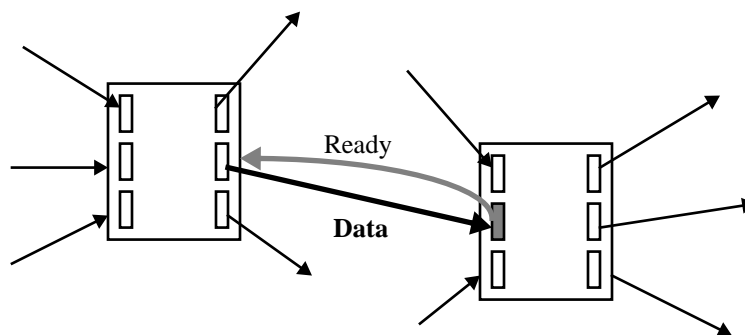
Figure 10-32 Link level flow-control

As a result of circumstances at the destination node, the storage at the input port may not be available to accept the data transfer, so the data must be retained at the source until the destination is ready

The implementation of link-level flow control differs depending on the design of the link, but the main idea is the same. The destination node provides feedback to the source indicating whether it is able to receive additional data on the link. The source holds onto the data until the destination indicates that it is able. Before we examine how this feedback is incorporated into the switch operation, we look at how the flow control is implemented on different kinds of links.

With short-wide links, the transfer across the link is essentially like a register transfer within a machine, extended with a couple of control signals. One may view the source and destination registers as being extended with a full/empty bit, as illustrated in Figure 10-33. If the source is full and the destination is empty, the transfer occurs, the destination become full, and the source becomes empty (unless it is refilled from its source). With synchronous operation (e.g., in the Cray T3D, IBM SP-2, TMC CM-5 and MIT J-machine) the flow control determines whether a transfer occurs for the clock cycle. It is easy to see how this is realized with edge-triggered or multi-phase level-sensitive designs. If the switches operate asynchronously, the behavior is much like a register transfer in a self-timed design. The source asserts the request signal when it is full and ready to transfer, the destination uses this signal to accept the value (when the input port is available) and asserts ack when it has accepted the data. With short-thin links the behavior is similar, excepts that series of phits is transferred for each req/ack handshake. .
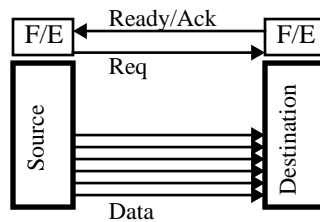
Figure 10-33 Simple Link level handshake

The source asserts its request when it has a flit to transmit, the destination acknowledges the receipt of a flit when it is ready to accept the next one. Until that time, the source repeatedly transmits the flit.

The req/ack handshake can be viewed as the transfer of a single token or credit between the source and destination. When the destination frees the input buffer, it passes the token to the source (i.e., increases its credit). The source uses this credit when it sends the next flit and must wait until its account is refilled. For long links, this credit scheme is expanded so that the entire pipeline associated with the link propagation delay can be filled. Suppose that the link is sufficiently long that several flits are in transit simultaneously. As indicated by Figure 10-34, it will also take several cycles for acks to propagate in the reverse direction, so a number of acks (credits) may be in transit as well. The obvious credit-based flow control is for the source to keep account of the available slots in the destination input buffer. The counter is initialized to the buffer size. It is decemented when a flit is sent and the output is blocked if the counter reaches zero. When the destination removes a flit from the input buffer, it returns a credit to the source, which increments the counter. The input buffer will never overflow; there is always room to drain the link into the buffer. This approach is most attractive with wide links, which have dedicated control lines for the reverse ack. For thin link, which multiplex ack symbols onto the opposite going channel, the ack per flit can be reduced by transferring bigger chunks of credit. However, there is still the problem that the approach is not very robust to loss of credit tokens.

Ideally, when the flows are moving forward smoothly there is no need for flow-control. The flow-control mechanism should be a governor which gently nudges the input rate to match the output rate. The propagation delays on the links give the system momentum. An alternative approach to link level flow control is to view the destination input buffer as a staging tank with a low water mark and a high water mark. When the fill level drops below the low mark, a GO symbol is sent
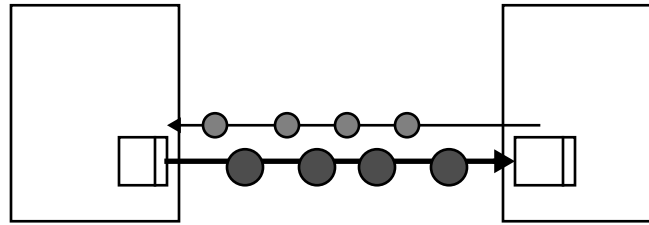
Figure 10-34 Transient Flits and Acks with long links.

With longer links the flow control scheme needs to allow more slack in order to keep the link full. Several flits can be driven onto the wire before waiting for acks to come back.

to the source, and when it goes over the high mark a STOP symbol is generated. There needs to be enough room below the low mark to withstand a full round-trip delay (the GO propagating to the source, being processed, and the first of a stream of flits propagating to the destination). Also, there must be enough headroom above the high mark to absorb the round-trip worth of flits that may be in-flight. A nice property of this approach is that redundant GO symbols may be sent anywhere below the high mark and STOP symbols may be sent anywhere above the low mark with no harmful effect. The fraction of the link bandwidth used by flow control symbols can be reduced by increasing the amount of storage between the low and high marks in the input buffer. This approach is used for example in Cal Tech routers[SeSu93] and the Myrinet commercial follow-on[Bod*95].
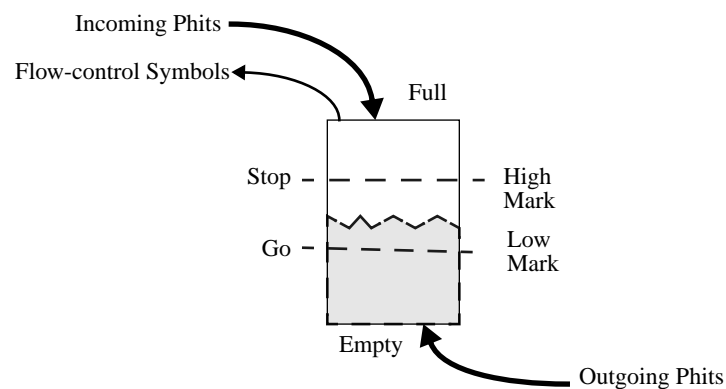
Figure 10-35 Slack Buffer Operation

When the fill level drops below the low mark, a GO symbol is sent to the source, and when it goes over the high mark a STOP symbol is generated.

It is worth noting that the link-level flow-control is used on host-switch links, as well as switch-switch links. In fact, it is generally carried over to the processor-NI interface as well. However, the techniques may vary for these different kinds of interfaces. For example, in the Intel Paragon the (175 MB/s) network links are all short with very small flit buffers. However, the NIs have

large input and output FIFOs. The communication assist includes a pair for DMA engines that can burst at 300 MB/s between memory and the network FIFOs. It is essential that the output (input) buffer not hit full (empty) in the middle of a burst, because holding the bus transaction would hurt performance and potential lead to deadlock. Thus, the burst is matched to the size of the middle region of the buffer and the high/low marks to the control turn-around between the NI and the DMA controller.

10.7.3 End-to-end flow control

Link-level flow-control exerts a certain amount of end-to-end control, because if congestion persists buffer will fill up and the flow will be controlled all the way back to the source host nodes. This is generally called *back-pressure*. For example, if k nodes are sending data to a single destination, they must eventually all slow to an average bandwidth of $1/k$-th the output bandwidth. If the switch scheduling is fair and all the routes through the network are symmetric, back-pressure will be enough to effect this. The problem is that by the time the sources feel the back-pressure and regulate their output flow, all the buffers in the tree from the hot spot to the sources are full. This will cause all traffic that crosses this tree to be slowed as well.

Hot-spots

This problem received quite a bit of attention as the technology reached a point where machines of several hundred to a thousand processors became reasonable.[PfNo85] If a thousand processors deliver on average any more than 0.1% of their traffic to any one destination, that destination becomes saturated. If this situation persists, as it might if frequent accesses are made to an important shared variable, a saturation tree forms in front of this destination that will eventually reach all the way to the sources. At this point, all the remaining traffic in the system is severely impeded. The problem is particularly pernicious in Butterflies, since there is only one route to each destination and there is a great deal of sharing among routes from one destination. Adaptive routing proves to make the hot spot problem even worse because traffic destined for the hot spot is sure to run into contention and be directed to alternate routes. Eventually the entire network clogs up. Large network buffers do not solve the problem, they just delay the onset. The time for the hot spot to clear is proportional to the total amount of hot spot traffic buffered in the network, so adaptivity and large buffers increase the time required for the hot spot to clear after the load is removed.

Various mechanisms have been developed to mitigate the causes of hot spots, such as all nodes incrementing a shared variable to perform a reduction or synchronization event. We examine these below, however, they only address situations where the problematic traffic is logically related. The more fundamental problem is that link-level flow control is like stopping on the freeway. Once the traffic jam forms, you are stuck. The better solution is not to get on the freeway at such times. With fewer packets inside the network, the normal mechanisms can do a better job of getting traffic to its destinations.

Global communication operations

Problems with simple back-pressure have been observed with completely balanced communication patterns, such as each node sending k packets to every other node. This occurs in many situations, including transposing a global matrix, converting between blocked and cyclic layouts, or the decimation step of an FFT [BrKu94,Dus*96]. Even if the topology is robust enough to avoid

serious internal bottlenecks on these operations (this is true of a fat-tree, but not a low-degree dimension-order mesh[Lei92]) a temporary backlog can have a cascading effect. When one destination falls behind in receiving packets from the network, a backlog begins to form. If priority is given to draining the network, this node will fall behind in sending to the other nodes. They in turn will send more than they receive and the backlog will tend to grow.

Simple end-to-end protocols in the global communication routines have been shown to mitigate this problem in practice. For example, a node may wait after sending a certain amount of data until it has also received this amount, or it may wait for chunks of its data to be acknowledged. These precautions keep the processors more closely in step and introduce small gaps in the traffic flow which decouples the processor-to-processor interaction through the network. (Interestingly, this technique is employed with the metering lights on the Oakland Bay bridge. Periodically, a wavefront of cars is injected into the bridge, separated by small gaps. This reduces the stochastic temporary blocking and avoids cascading blockage.)

Admission control

With shallow, cut-through networks the latency is low below saturation. Indeed in most modern parallel machine networks a single small message, or perhaps a few messages, will occupy an entire path from source to destination. If the remote network interface is not ready to accept the message, it is better to keep it within the source NI, rather than blocking traffic within the network. One approach to achieving this is to perform NI-to-NI credit-based flow-control. One study examining such techniques for a range of networks [CaGo95] indicates that allowing a single outstanding message per pair of NIs gives good throughput and maintains low latency.

10.8 Case Studies

Networks are a fascinating area of study from a practical design and engineering viewpoint because they do one simple operation – move information from one place to another – and yet there is a huge space of design alternatives. Although the most apparent characteristics of a network are its topology, link bandwidth, switching strategy, and routing algorithm, however, several more characteristics must be specified to completely describe the design. These include the cycle time, link width, switch buffer capacity and allocation strategy, routing mechanism, switch output selection algorithm, and the flow-control mechanism. Each component of the design can be understood and optimized in isolation, but they all interact in interesting ways to determine the global network performance and any particular traffic pattern, in the context of node architecture and the dependencies embedded in the programs running on the platform.

In this section, we summarize a set of concrete network design points in important commercial and research parallel architectures. Using the framework established above in this chapter, we systematically describe the key design parameters.

10.8.1 Cray T3D Network

The Cray T3D network consists of a 3-dimensional bidirectional torus of up to 1024 switch nodes, each connected directly to a pair of processors[1], with a data rate of 300 MB/s per channel. Each node is on a single board, along with two processors and memory. There are up to 128

nodes (256 processors) per cabinet; larger configurations are constructed by cabling together cabinets. Dimension-order, cut-through packet switched routing is used. The design of the network is very strongly influenced by the higher level design the rest of the system. Logically independent request and response networks are supported, with two virtual channels each to avoid deadlock, over a single set of physical links. Packets are variable length in multiples of 16 bits, as illustrated by Figure 10-36, and network transactions include various reads and writes, plus the ability to start a remote block-transfer engine (BLT), which is basically a DMA device. The first phit always contains the route, followed by the destination address, and the packet type or command code. The remainder of the payload depends on the packet type, consisting of relative addresses and data. All of the packet is parity protected, except the routing phit. If the route is corrupted it will be misrouted and the error will be detected at the destination because the destination address will not match the value in the packet.[1]
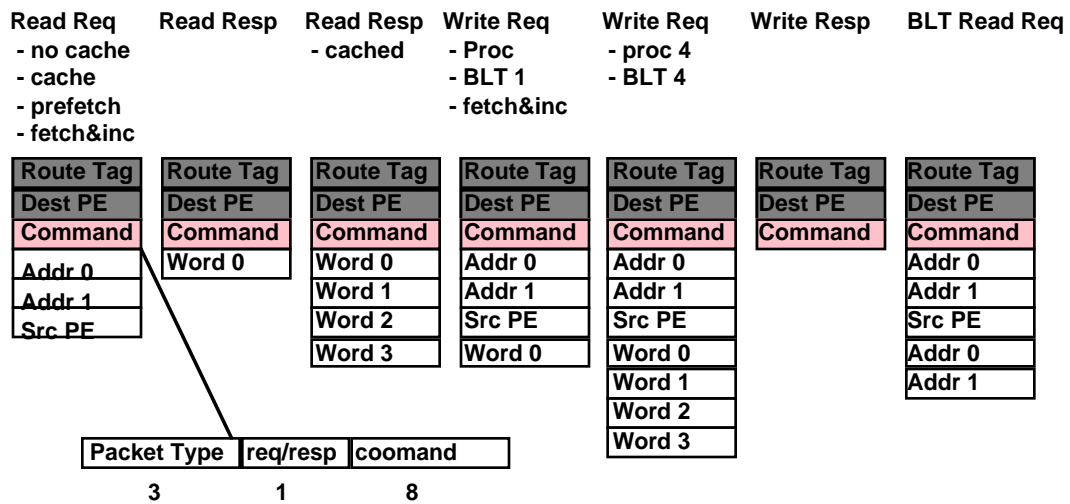
Figure 10-36 Cray T3D Packet Formats

All packets consist of a series of 16 bit phits, with the first three being the route and tag, the destination PE number, and the command. Parsing and processing of the rest of the packet is determined by the tag and command.

T3D links are short, wide, and synchronous. Each unidirectional link is 16 data and 8 control bits wide, operating under a single 150 MHz clock. Flits and phits are 16 bits. Two of the control bits identify the phit type (00 No info, 01 Routing tag, 10 packet, 11 last). The routing tag phit and the last-packet phit provide packet framing. Two additional control bits identify the virtual channel

---

1. Special I/O Gateway nodes attached to the boundary of the cube include two processors each attached to a 2-dimensional switch node connected into the X and Z dimensions. We will leave that aspect of the design and return to it in the I/O chapter.

1. This approach to error detection reveals a subtle aspect of networks. If the route is corrupted, there is a very small probability that it will take a wrong turn at just the wrong time and collide with another packet in a manner that causes deadlock within the network, in which case the end node error detection will not be engaged.

(req-hi, req-lo, resp-hi, resp-lo). The remaining four control lines are acknowledgments in the opposite direction, one per virtual channel. Thus, flow control per virtual channel and phits can be interleaved between virtual channels on a cycle by cycle basis.

The switch is constructed as three independent dimension routers, in six 10k gate arrays. There is modest amount of buffering in each switch, (8 16-bit parcels for each of four virtual channels in each of three dimensions), so packets compress into the switches when blocked. There is enough buffering in a switch to store small packets. The input port determines the desired output port by a simple arithmetic operation. The routing distance is decremented and if the result is non-zero the packet continues in its current direction on the current dimension, otherwise it is routed into the next dimension. Each output port uses a rotating priority among input ports requesting that output. For each input port, there is a rotating priority for virtual channels requesting that output.

The network interface contains eight packet buffers, two per virtual channel. The entire packet is buffered in the source NI before being transmitted into the network. It is buffered in the destination NI before being delivered to the processor or memory system. In addition to the main data communication network, separate tree-like networks for logical-AND (Barrier) and logical OR (Eureka) are provided.

The presence of bidirectional links provides two possible options in each dimension. A table lookup is performed in the source network interface to select the (deterministic) route consisting of the direction and distance in each of the three dimensions. An individual program occupies a partition of the machine consisting of a logically contiguous sub-array of arbitrary shape (under operating system configuration). A shift and mask logic within the communication assist maps the partition-relative virtual node address into a machine-wide logical <X,Y,Z> coordinate address. The machine can be configured with spare nodes, which can be brought in to replace a failed node. The <X,Y,Z> is used as an index for the route look-up, so the NI routing tables provides the final level of translation, identifying the physical node by its $<\pm x, \Delta x, \pm y, \Delta y, \pm z, \Delta z >$ route from the source. This routing lookup also identifies which of the four virtual channels is used. To avoid deadlock within either the request or response (virtual) network the high channel is used for packets that cross the dateline, and the low channel otherwise.

10.8.2 IBM SP-1, SP-2 Network

The network used in the IBM SP-1 and SP-2[Abay94, Stu*94] parallel machines is in some ways more versatile than that in the Cray T3D, but of lower performance and without support for two phase, request-response operation. It is packet switched, with cut-through, source-based routing and no virtual channels. The switch has eight bidirectional 40 MB/s ports and can support a wide variety of topologies. However, in the SP machines a collection of switches are packaged on a board as a 4-ary 2-dimensional butterfly with sixteen internal connections to hosts in the same rack as the switch board and sixteen external connections to other racks, as illustrated in Figure 10-37. The rack-level topology varies from machine to machine, but is typically a variant of a butterfly. Figure 1-23 of Chapter 1 shows the large IBM SP-2 configuration at the Maui High Performance Computing Center. Individual cabinets have the first level of routing at the bottom connecting the 16 internal nodes to 16 external links. Additional cabinets provide connectivity between collections of cabinets. The wiring for these additional levels is located beneath the machine room floor. Since the physical toplogy is not fixed in hardware, the network interface inserts the route for each outgoing message via a table lookup.
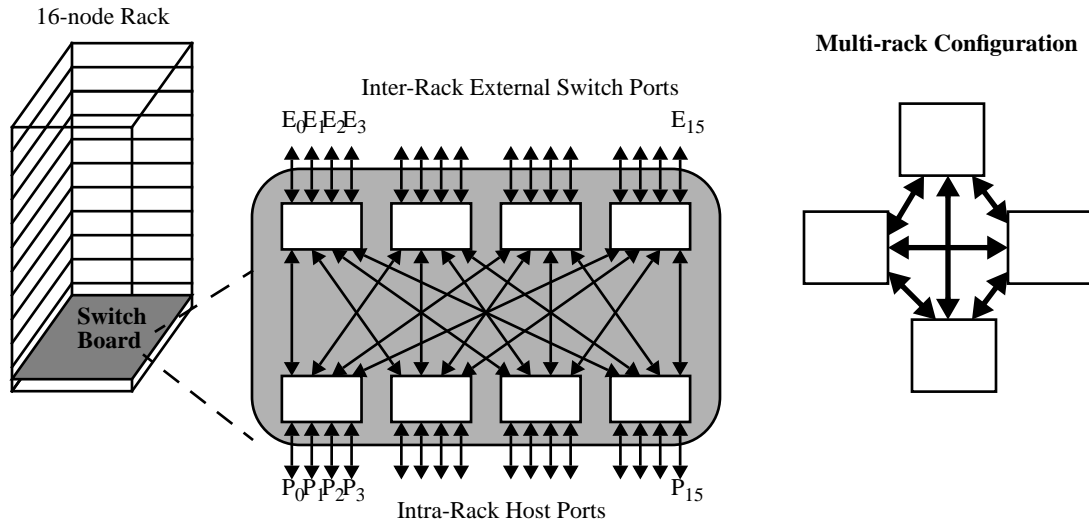
Figure 10-37 SP Switch Packaging

The collection of switches within a rack provide bidirectional connection between 16 internal ports and 16 external ports.

Packets consist of a sequence of up to 255 bytes, the first byte is the packet length, followed by one or more routing bytes and then the data bytes. Each routing byte contains two 3-bit output specifiers, with an additional selector bit. The links are synchronous, wide, and long. A single 40 MHz clock is distributed to all the switches, with each link tuned so that its delay is an integral number of cycles. (Interboard signals are 100K ECL differential pairs. On-board clock trees are also 100K ECL.) The links consist of ten wires: eight data bits, a framing "tag" control bit, and a reverse flow-control bit. Thus, the phit is one byte. The tag bit identifies the length and routing phits. The flit is two bytes; two cycles are used to signal the availability of two bytes of storage in the receiver buffer. At any time, a stream of data/tag flits can be propagating down the link, while a stream of credit tokens propagate in the other direction.

The switch provides 31 bytes of FIFO buffering on each input port, allowing links to be 16 phits long. In addition, there are 7 bytes of FIFO buffering on each output and a shared central queue holding 128 8-byte chunks. As illustrated in Figure 10-38, the switch contains both an unbuffered byte-serial cross-bar and a 128 x 64-bit dual port RAM as interconnect between input and output port. After two bytes of the packet have arrived in the input port, the input port control logic can request the desired output. If this output is available, the packet cuts through via the cross-bar to the output port, with a minimum routing delay of 5 cycles per switch. If the output port is not available, the packet fills into the input FIFO. If the output port remains blocked, the packet is spooled into the central queue in 8-byte "chunks." Since the central queue accepts one 8-byte input and one 8-byte output per cycle, its bandwidth matches that of the eight byte serial input and output ports of the switch. Internally, the central queue is organized as eight FIFO linked lists, one per output port, using and auxiliary 128x7-bit RAM for the links. One 8-byte chunk reserved for each output port. Thus, the switch operates in byte serial mode when the load is low,

but when contention forms it time-multiplexes 8-byte chunks through the central queue, with the inputs acting as a deserializer and the outputs as a serializer.
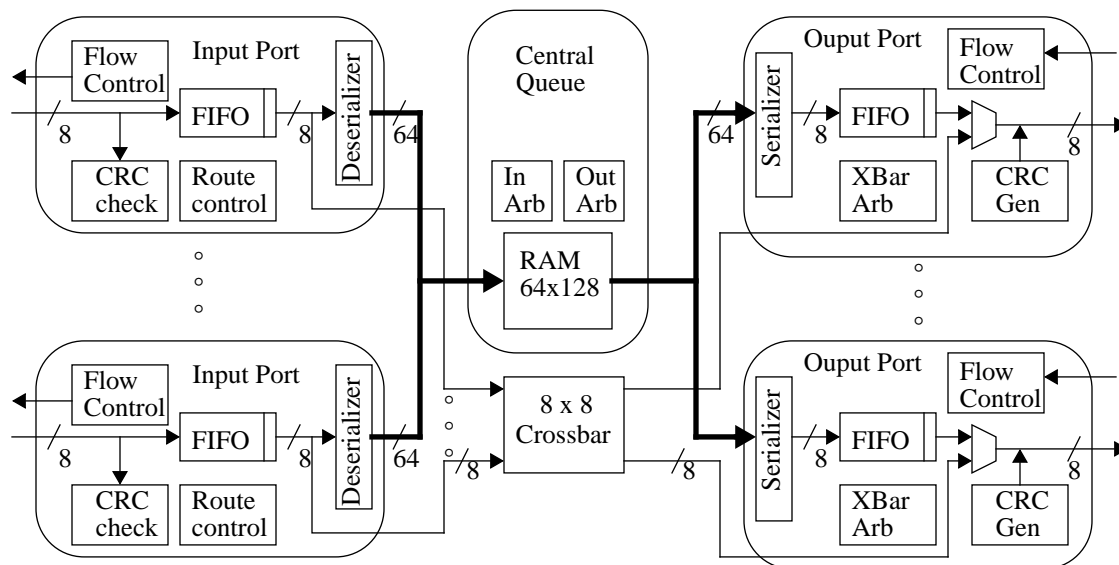
Figure 10-38 IBM SP (Vulcan) switch design

The IBM SP switch uses a cross-bar for unbuffered packets passed directly from input port to output port, but all buffered packets are shunted into a common memory pool.

Each output port arbitrates among requests on an LRU basis, with chunks in the central queue having priority over bytes in input FIFOs. Output ports are served by the central queue in LRU order. The central queue gives priority to inputs which chunks destined for unblocked output ports.

The SP network has three unusual aspects. First, since the operation is globally synchronous, instead of including CRC information in the envelop of each packet, time is divided into 64-cycle "frames." The last two phits of each frame carry CRC. The input port checks the CRC and the output port generates it (after stripping of used routing phits). Secondly, the switch is a single chip and every switch chip is shadowed by an identical switch. The pins are bidirectional I/O pins, so one of the chips merely checks the operation of the other. (This will detect switch errors, but not link errors.) Finally, the switch supports a circuit switched "service mode" for various diagnostic purposes. The network is drained free of packets before changing modes.

10.8.3 Scalable Coherent Interconnect

The Scalable Coherent Interface provides a well specified case study in high performance interconnects, because it emerged through a standards process, rather than as a proprietary design or academic proposal. It was a standard long time in coming, but has gained popularity as implementations have got underway. It has been adopted by several vendors, although in many cases only a portion of the specification is followed. Essentially the full SCI specification is used in the

interconnect of the HP/Convex Exemplar and in the Sequent NUMA-Q. The Cray SCX I/O network is based heavily on SCI.

A key element of the SCI design is that it builds around the concept of unidirectional rings, called ringlets, rather than bidirectional links. Ringlets are connected together by switches to form large networks. The specification defines three layers. A physical layer, a packet layer, and a transaction layer. The physical layer is specified in two 1 GB/s forms, as explained in Example . Packets consist of a sequence of 16 bit units, much like the Cray T3D packets. As illustrated in Figure 10-39, all packets consists of a TargetId, Command, and SourceId, followed by zero or more command specific units, and finally a 32-bit CRC. One unusual aspect, is that source and target Ids are arbitrary 16 bit node addresses. The packet does not contain a route. In this sense, the design is like a bus. The source puts the target address on the interconnect and the interconnect determines how to get the information to the right place. Within a ring this is simple, because the packet circulates the ring and the target extracts the packet. In the general case, switches use table driven routing to move packets from ring to ring.

General Packet Format

| target id |
| command |
| source id |
| control |
| Address - Status (0or 6 bytes) |
| ext (0-16 bytes) |
| data (0 or 16 bytes) |
| CRC |

request

| mdr | spr | phase | old | ech0 | eh | cmd |
| 2 | 2 | 2 | 1 | 1 | 1 | 7 |

request echo

| mdr | spr | phase | old | ech1 | bsy | res0 | trans id |
| 2 | 2 | 2 | 1 | 1 | 1 | 1 | 6 |

response

| mdr | spr | phase | old | ech0 | eh | cmd |
| 2 | 2 | 2 | 1 | 1 | 1 | 7 |

response echo

| mdr | spr | phase | old | ech1 | bsy | res1 | trans id |
| 2 | 2 | 2 | 1 | 1 | 1 | 1 | 6 |

control

| trace | tod Exponent | tod Mantissa | tpr | trans id |
| 2 | 5 | 2 | 2 | 6 |

Figure 10-39 SCI Packet formats

SCI operations involve a pair transactions, a request and a response. Owing to its ring-based underpinnings, each transaction involves conveying a packet from the source to the target and an echo (going the rest of the way around the ring) from the target back to the source. All packets are a sequence of 16-bit units, with the first three being the destination node, command, and source node, and the final being the CRC. Requests contain a 6 byte address, optional extension, and optional data. Responses have a similar format, but the address bytes carry status information. Both kinds of echo packets contain only the minimal four units. The command unit identifies the packet type through the phase and ech fields. It also describes the operation to be performed (request cmd) or matched (trans id). The remaining fields and the control unit address lower level issues of how packet queuing and retry are handled at the packet layer.

An SCI transaction, such as read or write, consists of two network transactions (request and response), each of which has two phases on each ringlet. Let's take this step by step. The source node issues a request packet for a target. The request packet circulates on the source ringlet. If the target is on the same ring as the source, the target extracts the request from the ring and replaces it with a echo packet, which continues around the ring back to the source, whereupon it is removed from the ring. The echo packet serves to inform the source that the original packet was either accepted by the target, or rejected, in which the echo contains a NAK. It may be rejected

either because of a buffer full condition or because the packet was corrupted. The source maintains a timer on outstanding packets, to it can detect if the echo gets corrupts. If the target is not on the source ring, a switch node on the ring serves as a proxy for the target. It accepts the packet and provides the echo, once it has successfully buffered the packet. Thus, the echo only tells the source that the packet successfully left the ringlet. The switch will then initiate the packet onto another ring along the route to the target. Upon receiving a request, the target node will initiate a response transaction. It too will have a packet phase and an echo phase on each ringlet on the route back to the original source. The request echo packet informs the source of the transaction ide assigned to the target; this is used to match the eventual response, much as on a split-phase bus.

Rather then have a clean envelop for each of the layers, in SCI they blur together a bit in the packet format where several fields control queue management and retry mechanisms. The control tpr (transaction priority), command mpr (maximum ring priority) and command spr (send priority) together determine one of four priority levels for the packet. The transaction priority is initiallly set by the requestor, but the actual send priority is established by nodes along the route based on what other blocked transactions they have in their queues. The phase and busy fields are used as part of the flow control negotiation between the source and target nodes.

In the Sequent NUMA-Q, the 18-bit wide SCI ring is driven directly by the DataPump in a Quad at 1GB/sec node-to-network bandwidth. The transport layer follows a strict request-reply protocol. When the DataPump puts a packet on the ring, it keeps a copy of the packet in its outgoing buffer until an acknowledgment (called an *echo*) is returned. When a DataPump removes an incoming packet from the ring it replaces it by a *positive echo* that is returned to the sender along the ring. If a DataPump detects a packet destined for it but does not have space to remove that packet from the ring, it sends a *negative echo* (like a NACK), which causes the sender to retry its transmission (still holding the space in its outgoing buffer).

Since the latency of communication on a ring increases linearly with the number of nodes on it, large high-performance SCI systems are expected to be built out of smaller rings interconnected in arbitrary network topologies. For example, a performance study indicates that a single ring can effectively support about 4-8 high-performance processing nodes [SVG92]. The interconnections among rings are expected to be either multiple small *bridges*, each connecting together a small number of rings, or centralized *switches* each connecting a large number of rings.

10.8.4 SGI Origin Network

The SGI Origin network is based on a flexible switch, called SPIDER, that supports six pairs of unidirectional links, each pair providing over 1.56 GB/s of total bandwidth in the two directions. Two nodes (four processors) are connected to each router, so there are four pairs of links to connect to other routers. This building block is configured in a family of topologies related to hypercunes, as illustrated in Figure 10-40. The links are flexible (long, wide) cables that can be up to 3 meters long. The peak bisection bandwidth of a maximum configuration is over 16GB/s. Messages are pipelined through the network, and the latency through the router itself is 41ns from pin to pin. Routing is table driven, so as part of the network initialization the routing tables are set up in each switch. This allows routing to be programmable, so it is possible to support a range of configurations, to have a partially populated network (i.e., not all ports are used), and route around faulty components. Availability is also aided by the fact that the routers are separately powered, and provide per-link error protection with hardware retry upon error. The switch pro-

vides separate virtual channels for requests and replies, and supports 256 levels of message priority, with packet aging.

| (a) 2-node | (b) 4-node | (c) 8-node | (d) 16-node | (d) 32-node |

(e) 64-node
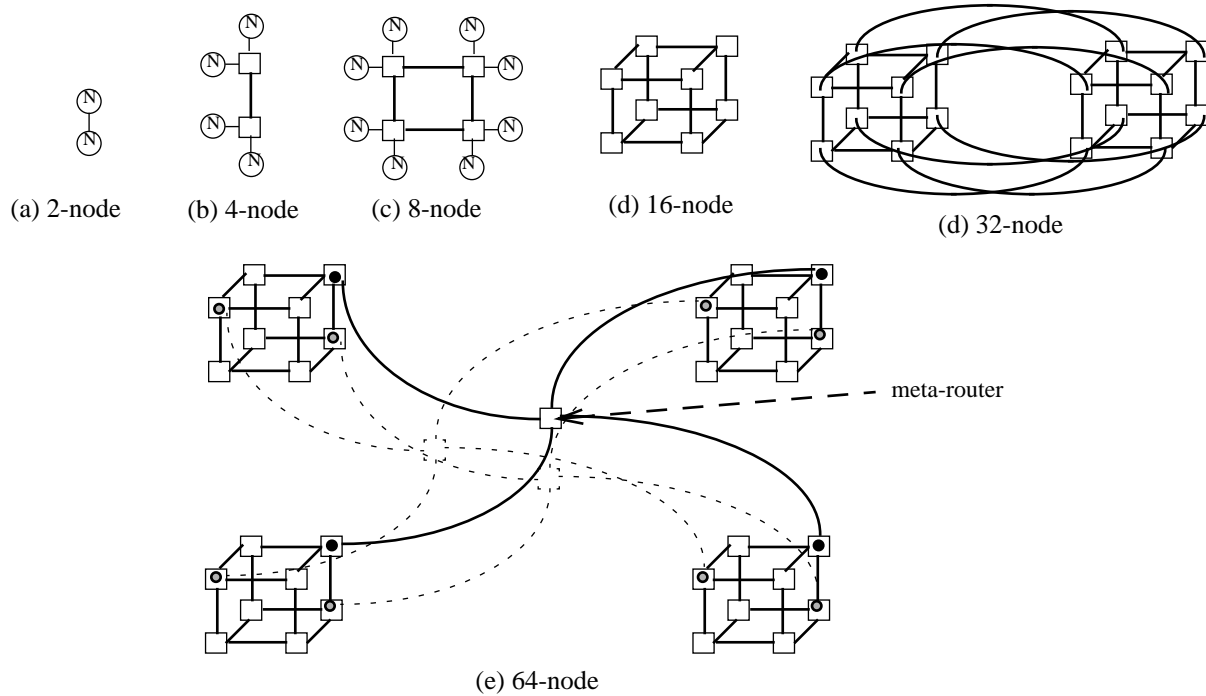
Figure 10-40 Network topology and router connections in the Origin multiprocessor.

Figures (d) and (e) show only the routers and omit the two nodes connected to each for simplicity. Figure (e) also shows only a few of the fat-cube connections across hypercubes; The routers that connect 16-node subcubes are called meta-routers. For a 512-node (1024-processor) configuration, each meta-router itself would be replaced by a 5-d router hypercube.

10.8.5 Myricom Network

As final case study, we briefly examine the Myricom network [Bod95] used in several cluster systems. The communication assist within the network interface card was described in Section 7.8. Here we are concerned with the switch that is used to construct a scalable interconnection network. Perhaps the most interesting aspect of its design is it simplicity. The basic building block is a switch with eight bidirectional ports of 160 MB/s each. The physical link in a 9 bit wide, long wire. It can be up to 25 m and 23 phits can be in transit simultaneously. The encoding allows control flow, framing symbols, and 'gap' (or idle) symbols to be interleaved with data symbols. Symbols are constantly transmitted between switches across the links, so the switch can determine which of its ports are connected. A packet is simply a sequence of routing bytes followed by a sequence of payload bytes and a CRC byte. The end of a packet is delimited by the presence of a gap symbol. The start of a packet is indicated by the presence of a non-gap symbol.

The operation of the switch is extremely simple, it removes the first byte from an incoming packet, computes the output port by adding the contents of the route byte to the input port number and spools the rest of the packet to that port. The switch uses wormhole routing with a small amount of buffering for each input and each output. The routing delay through a switch is less

than 500 ns. The switches can be wired together in an arbitrary topology. It is the responsibility of the host communication software to construct routes that are valid for the physical interconnection of the network. If a packet attempts to exit a switch on an invalid or unconnected port, it is discarded. When all the routing bytes are used up, the packet should have arrived at a NIC and the first byte of the message has a bit to indicate that it is not a routing byte and the remaing bits indicate the packet type. All higher level packet formating and transactions are realized by the NIC and the host, the interconnection network only moves the bits.

10.9 Concluding Remarks

Parallel computer networks present a rich and diverse design space that brings together several levels of design. The physical layer, link level issues represent some of the most interesting electrical engineering aspects of computer design. In a constant effort to keep pace with the increasing rate of processors, link rates are constantly improving. Currently we are seeing multigigabit rates on copper pairs and parallel fiber technologies offer the possibility of multigigabyte rates per link in the near future. One of the major issues at these rates is dealing with errors. If bit error rates of the physical medium are in the order of 10^{-19} per bit and data is being transmitted at a rate of 10^9 bytes per second, then an error is likely to occur on a link roughly every 10 minutes. With thousands of links in the machine, errors occur every second. These must be detected and corrected rapidly.

The switch to switch layer of design also offers a rich set of trade-offs, including how aspects of the problem are pushed down into the physical layer, and how they are pushed up into the packet layer. For example, flow control can be built in to the exchange of digital symbols across a link, or it can be addressed at the next layer in terms of packets thar cross the link, or even one layer higher, in terms of messages sent from end to end. There is a huge space of alternatives for the design of the switch itself, and these are again driven by engineering constraints from below and design requirements from above.

Even the basic topology, switching strategy, and routing algorithm reflects a compromise of engineering requirements and design requirements from below and above. We have seen, for example, how a basic question like the degree of the switch depends heavily on the cost model associated with the target technology and the characteristics of the communication pattern in the target workload. Finally, as with many other aspects of architecture, there is significant room for debate as to how much of the higher level semantics of the node-to-node communication abstraction should be embedded in the hardware design of the network itself. The entire area of network design for parallel computers is bound to be exciting for many years to come. In particular, there is a strong flow of ideas and technology between parallel computer networks and advancing, scalable networks for local area and system area communication.

10.10 References

[AbAy94]   B. Abali and C. Aykanat, Routing Algorithms for IBM SP1, Lecture Notes in Comp. Sci, Springer-Verlag, vol. 853, 1994 pp. 161-175.

[Aga91]   A. Agarwal, Limit on Interconnection Performance, IEEE Transactions on Parallel and Distributed Systems, V. 2, No. 4, Oct. 1991, pp. 398-412.

[And*92]   T. E. Anderson, S. S. Owicki, J. P. Saxe, C. P. Thacker, High Speed Switch Scheduling for Local Area Networks, Proc. of ASPLOS V, pp. 98-110, oct. 1992.

[Arn*89]   E. A. Arnold, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. Steenkiste, The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers, Proc. of ASPLOS III, Apr. 1989, pp. 205-216.

[Bak90]   H. B. Bakoglu, Circuits, Interconnection, and Packaging for VLSI, Addison-Wesley Publishing Co., 1990.

[Bak*95]   W. E. Baker, R. W. Horst, D. P. Sonnier, and W. J. Watson, A Flexible ServerNet-based Fault-Tolerant Architecture, 25th Annual Symposium on Fault-Tolerant Computing, Jun. 1995.

[Ben65]   V. Benes. Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press,. New York, NY, 1965.

[BhLe84]   S. Bhatt and C. L. Leiserson, How to Assemble Tree Machines, Advances in Computing Research, vol. 2, pp95-114, JAI Press Inc, 1984.

[Bod*95]   N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su, Myrinet: A Gigabit-per-Second Local Area Network, IEEE Micro, Feb. 1995, vol.15, (no.1), pp. 29-38.

[Bor*90]   S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iwarp.In Proc. 17th Intl. Symposium on Computer Architecture, pages 70-81. ACM, May 1990. Revised version appears as technical report CMU-CS-90-197.

[Bre*94]   E. A. Brewer, F. T. Chong, F. T. Leighton, Scalable Expanders: Exploiting Hierarchical Random Wiring, Proc. of the 1994 Symp. on the Theory of Computing, Montreal Canada, May, 1994.

[BrKu94]   Eric A. Brewer and Bradley C. Kuszmaul, How to Get Good Performance from the CM-5 Data Network, Proceedings of the 1994 International Parallel Processing Symposium, Cancun, Mexico, April 1994.

[CaGo95]   T. Callahan and S. C. Goldstein, NIFDY: A Low Overhead, High Throughput Network Interface, Proc. of the 22nd Annual Symp. on Computer Architecture, June 1995.

[ChKi92]   A. A. Chien and J. H. Kim, Planar-Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors, Proc. of ISCA, 1992.

[Coh*93]   D. Cohen, G. Finn, R. Felderman, and A. DeSchon, ATOMIC: A Low-Cost, Very High-Speed, Local Communication Architecture, Proc. of the 1993 Int. Conf. on Parallel Processing, 1993.

[Dal90a]   William J. Dally, Virtual-Channel Flow Control, Proc. of ISCA 1990, Seattle WA.

[Dal90b]   William J. Dally, Performance Analysis of k-ary n-cube Interconnection Networks, IEEE-TOC, V. 39, No. 6, June 1990, pp. 775-85.

[DaSe87]   W. Dally and C. Seitz, Deadlock-Free Message Routing in Multiprocessor Interconnections Networks, IEEE-TOC, vol c-36, no. 5, may 1987.

[Dus*96]   Andrea C. Dusseau, David E. Culler, Klaus E. Schauser, and Richard P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. IEEE Transactions on Parallel and Distributed Systems, 7(8):791-805, August 1996.

[Fel94]   R. Felderman, et al, "Atomic: A High Speed Local Communication Architecture", J. of High Speed Networks, Vol. 3, No. 1, pp.1-29, 1994.

[GlNi92]   C. J. Glass, L. M. Ni, The Turn Model for Adaptive Routing, Proc. of ISCA 1992. pp. 278-287.

[GrLe89]   R. I. Greenberg and C. E. Leiserson, Randomized Routing on Fat-Trees, Advances in Computing Research, vol. 5, pp. 345-374, JAI Press, 1989.

[Gro92]   W. Groscup, The Intel Paragon XP/S supercomputer. Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Nov 1992, pp. 262--273.

[Gun81]   K. D. Gunther, Prevention of Deadlocks in Packet-Switched Data Transport Systems, IEEE Transactions on Communication, vol. com-29, no. 4, Apr. 1981, pp. 512-24.

[HiTu93]   W. D. Hillis and L. W. Tucker, The CM-5 Connection Machine: A Scalable Supercomputer, Communications of the ACM, vol. 36, no. 11, nov. 1993, pp. 31-40.

[HoMc93]   Mark Homewood and Moray McLaren, Meiko CS-2 Interconnect Elan - Elite Design, Hot Interconnects, Aug. 1993.

[Hor95]   R. Horst, TNet: A Reliable System Area Network, IEEE Micro, Feb. 1995, vol.15, (no.1), pp. 37-45.

[Joe94]   C. F. Joerg, Design and Implementation of a Packet Switched Routing Chip, Technical Report, MIT Laboratory for Computer Science, MIT/LCS/TR-482, Aug. 1994.

[JoBo91]   Christopher F. Joerg and Andy Boughton, The Monsoon Interconnection Network, Proc. of ICCD, Oct. 1991.

[Kar*87]   M. Karol, M. Hluchyj, S. Morgan, Input versus Output Queueing on a Space Division Packet Switch, IEEE Transactions on Communications, v. 35, no. 12, pp. 1347-56, Dec. 1987.

[Kar*90]   R. Karp, U. Vazirani, V. Vazirani, An Optimal Algorithm for on-line Bipartite Matching, Proc. of the 22dn ACM Symposium on the Theory of Computing, pp. 352-58, May, 1990.

[KeSc93]   R. E. Kessler and J. L. Schwarzmeier, Cray T3D: a new dimension for Cray Research, Proc. of Papers. COMPCON Spring '93, pp 176-82, San Francisco, CA, Feb. 1993.

[KeKl79]   P. Kermani and L. Kleinrock, Virtual Cut-through: A New Computer Communication Switching Technique, Computer Networks, vol. 3., pp. 267-286, sep. 1979.

[Koe*94]   R. Kent Koeninger, Mark Furtney, and Martin Walker. A Shared Memory MPP from Cray Research, Digital Technical Journal 6(2):8-21, Spring 1994.

[KoSn]   S. Kostantantindou and L. Snyder, Chaos Router: architecture and performance, Proc. of the 18th Annual Symp. on Computer Architecture, pp. 212-23, 1991.

[KrSn83]   C. P. Kruskal and M. Snir, The Perormance of Multistage Interconnection Networks for Multiprocvessors, IEEE Transactions on Computers, vol. c-32, no. 12, dec. 1983, pp. 1091-1098.

[Kun*91]   Kung, H.T, et al, Network-based multicomputers: an emerging parallel architecture, Proceedings Supercomputing '91, Albuquerque, NM, USA, 18-22 Nov. 1991 p. 664-73

[Leis85]   C. E. Leiserson, Fat-trees: Universal Networks for Hardware-Efficient Supercomputing, IEEE Transactions on Computers, Vol. c-34, no. 10, Oct. 1985, pp. 892-901

[Lei92]   F. Thomson Leighton, Introduction to Parallel Algorithms and Architectures, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1992.

[Lei*92]   C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmau, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, R. Zak. The Network Architecture of the CM-5. Symposium on Parallel and Distributed Algorithms '92" Jun 1992, pp. 272-285.

[Li88]   Li, S-Y, Theory of Periodic Contention and Its Application to Packet Switching, Proc. of INFOCOM '88, pp. 320-5, mar. 1988.

[LiHa91]   D. Linder and J. Harden, An Adaptive Fault Tolerant Wormhole strategy for k-ary n-cubes, IEEE Transactions on Computer, C-40(1):2-12, Jan., 1991.

[Mai*97]   A. Mainwaring, B. Chun, S. Schleimer, D. Wilkerson, System Area Network Mapping, Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architecture, Newport, RI, June 1997, pp.116-126.

[NuDa92]   P. Nuth and W. J. Dally, The J-Machine Network, Proc. of International Conference on Computer Design: VLSI in computers and processors, Oct. 1992.

[NgSe89]   J. Ngai and C. Seitz, A Framework for Adaptive Routing in Multicomputer Networks, Proc. of the 1989 Symp. on Parallel Algorithms and Architectures, 1989.

[PeDa96]   L. Peterson and B. Davie, Computer Networks, Morgan Kaufmann Publishers, Inc., 1996.

[Pfi*85]   G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliff, E. A. Melton, V. A. Norton, and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, International Conference on Parallel Processing, 1985.

[PfNo85]   G. F. Pfister and V. A. Norton, "Hot Spot" COntention and Combining Multistage Interconnection Networks, IEEE Transactions on Computers, vol. c-34, no. 10, oct. 1985.

[Rat85]   J. Ratner, Concurrent Processing: a new direction in scientific computing, Conf. Proc. of the 1985 Natinal Computing Conference, p. 835, 1985.

[Ret*90]   R. Rettberg, W. Crowther, P. Carvey, and R. Tomlinson, The Monarch Parallel Processor Hardware Design, IEEE Computer, April 1990, pp. 18-30.

[ReTh96]   R. Rettberg and R. Thomas, Contention is no Obstacle to Shared-Memory Multiprocessing, Communications of the ACM, 29:12, pp. 1202-1212, Dec. 1986.

[ScGo94]   S. L. Scott and J. R. Goodman, The Impact of Pipelined Channels on k-ary n-Cube Networks, IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 1, jan. 1994, pp. 2-16.

[Sch*90]   M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, C. P. Thacker, Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links, IEEE Journal on Selected Areas in Communications, vol. 9, no. 8. pp. 1318-35, oct. 1991.

[ScJa95]   L. Schwiebert and D. N. Jayasimha, A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks, SPAA, Jul 1995, pp. 175-84.

[Sei85]   Charles L. Seitz, The Cosmic Cube, Communications of the ACM, 28(1):22-33, Jan 1985.

[SeSu93]   Charles L/ Seitz and Wen-King Su, A Family of Routing and Communication Chips Based on Mosaic, Proc of Univ. of Washington Symposium on Integrated Systems, MIT PRess, pp. 320-337.

[SP2]   C. B. Stunkel *et al.* The SP2 Communication Subsystem. On-line as `http://ibm.tc.cornell.edu/ibm/pps/doc/css/css.ps`.

[Stu*94]   C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao, The SP-1 High Performance Switch, Proc. of the Scalable High Performance Computing Conference, Knoxville, TN pp. 150-57, May 1994.

[SVG92]   Steven Scott, Mary Vernon and James Goodman. Performance of the SCI Ring. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 403-414, May 1992.

[TaFr88]   Y. Tamir and G. L. Frazier, High-performance Multi-queue Buffers for VLSI Communication Switches, 15th Annual International Symposium on Computer Architecture, 1988, pp. 343-354.

[TrDu92]   R. Traylor and D. Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. *Proc. of Hot Chips '92 Symposium*, August 1992.

[Tur88]   J. S. Turner, Design of a Broadcast Packet Switching Network, IEEE Transactions on Communication, vol 36, no. 6, pp. 734-43, June, 1988.

[vEi*92]   von Eicken, T.; Culler, D.E.; Goldstein, S.C.; Schauser, K.E., Active messages: a mechanism for integrated communication and computation, Proc. of 19th Annual International Symposium on Computer Architecture, old Coast, Qld., Australia, 19-21 May 1992, pp. 256-66

10.11 Exercises

10.1 Consider a packet format with 10 bytes of routing and control information and 6 bytes of CRC and other trailer information. The payload contains a 64 byte cache block, along with 8 bytes of command and address information. If the raw link bandwidth is 500 MB/s, what is the effective data bandwidth on cache block transfers using this format? How would this change with 32 byte cache blocks? 128 byte blocks? 4 KB page transfers?

10.2 Suppose the links are 1 byte wide and operating at 300 MHz in a network where the average routing distance between nodes is $\log_4 P$ for P nodes. Compare the unloaded latency for 80 byte packets under store-and-forward and cut-through routing, assuming 4 cycles of delay per hop to make the routing decision and P ranging from 16 to 1024 nodes.

10.3 Perform the comparison in Exercise 10.2 for 32 KB transfers fragmented into 1 KB packets.

10.4 Find an optimal fragmentation strategy for a 8KB page sized message under the following assumptions. There is a fixed overhead of o cycles per fragment, there is a store-and-forward delay through the source and the destination NI, packets cut-through the network with a routing delay of R cycles, and the limiting data transfer rate is b bytes per cycle.

10.5 Suppose an $n \times n$ matrix of double precision numbers is laid out over P nodes by rows, and in order to compute on the columns you desire to transpose it to have a column layout. How much data will cross the bisection during this operation?

10.6 Consider a 2D torus direct network of N nodes with a link bandwidth of b bytes per second. Calculate the bisection bandwidth and the average routing distance. Compare the estimate of aggregate communication bandwidth using Equation 10.6 with the estimate based on bisection. In addition, suppose that every node communicates only with nodes 2 hops away. Then what bandwidth is available? What is each node communicates only with nodes in its row?

10.7 Show how an N node torus is embedded in an N node hypercube such that neighbors in the torus, i.e., nodes of distance one, are neighbors in the hypercube. (Hint: observe what happens when the addresses are ordered in a greycode sequence.) Generalize this embedding for higher dimensional meshes.

10.8 Perform the analysis of Equation 10.6 for a hypercube network. In the last step, treat the row as being defined by the greycode mapping of the grid into the hypercube.

10.9 Calculate the average distance for a linear array of N nodes (assume N is even) and for a 2D mesh and 2D torus of N nodes.

10.10 A more accurate estimate of the communication time can be obtained by determining the average number of channels traversed per packet, h_{ave}, for the workload of interest and the particular network topology. The effective aggregate bandwidth is at most $\frac{\|C\|}{h_{ave}} B$. Suppose the orchestration of a program treats the processors as a $\sqrt{p} \times \sqrt{p}$ logical mesh where each node communicates n bytes of data with its eight neighbors in the four directions and on the diagonal. Give an estimate of the communication time on a grid and a hypercube.

10.11 show how an N node tree can be embedded in an N node hypercube by stretching one of the tree edges across two hypercube edges.

10.12 Use a spread sheet to construct the comparison in Figure 10-12 for three distinct design points: 10 cycle routing delays, 1024 byte messages, and 16-bit wide links. What conclusions can you draw?

10.13 Verify that the minimum latency is achieved under equal pin scaling in Figure 10-14 when the routing delay is equal to the channel time.

10.14 Derive a formula for dimension that achieves the minimum latency is achieved under equal bisection scaling based on Equation 10.7.

10.15 Under the equal bisection scaling rule, how wide are the links of a 2D mesh with 1 million nodes?

10.16 Compare the behavior under load of 2D and 3D cubes of roughly equal size, as in Figure 10-17, for equal pin and equal bisection scaling.

10.17 Specify the boolean logic used to compute the routing function in the switch with Δx, Δy routing on 2D mesh.

10.18 If Δx, Δy routing is used, what effect does decrementing the count in the header have on the CRC in the trailer? How can this issue be addressed in the switch design?

10.19 Specify the boolean logic used to compute the routing function in the switch dimension order routing on a hypercube.

10.20 Prove the e-cube routing in a hypercube is deadlock free.

10.21 Construct the table-based equivalent of $[\Delta x, \Delta y]$ routing in an 2D mesh.

10.22 Show that permitting the pair of turns not allowed in Figure 10-24 leaves complex cycles. hint: They look like a figure 8.

10.23 Show that with two virtual channels, arbitrary routing in a bidirectional network can be made deadlock free.

10.24 (Pick any flow control scheme in the chapter and compute fraction of bandwidth used by flow control symbols.

10.25 Revise latency estimates of Figure 10-14 for stacked dimension switches.

10.26 Table 10-1 provides the topologies and an estimate of the basic communication performance characteristics for several important designs. For each, calculate the network latency for 40 byte and 160 byte messages on 16 and 1024 node configurations. What fraction of this is routing delay and what fraction is occupancy?

Table 10-1 Link width and routing delay for various parallel machine networks

Machine	Topology	Cycle Time (ns)	Channel Width (bits)	Routing Delay (cycles)	FLIT (data bits)
nCUBE/2	Hypercube	25	1	40	32
TMC CM-5	Fat tree	25	4	10	4
IBM SP-2	Banyan	25	8		
Intel Paragon	2d Mesh	11.5	16	2	16
Meiko CS-2	Fat tree	20	8		
Cray T3D	3D torus	6.67	16		16
Dash	Torus	30	16		
J-Machine	3d Mesh	31	8	1	8
Monsoon	Butterfly	20	16		

CHAPTER 11   Latency Tolerance

11.1 Introduction

In Chapter 1, we saw that while the speed of microprocessors increases by more than a factor of ten per decade, the speed of commodity memories (DRAMs) only doubles, i.e., access time is halved. Thus, the latency of memory access in terms of processor clock cycles grows by a factor of six in ten years! Multiprocessors greatly exacerbate the problem. In bus-based systems, the imposition of a high-bandwidth bus between processor and memory tends to increase the latency of obtaining data from memory: Recall that in both the SGI Challenge and the Sun Enterprise about a third of the read miss penalty was spent in the bus protocol. When memory is physically distributed, the latency of the network and the network interface is added to that of accessing the local memory on the node. Caches help reduce the frequency of high-latency accesses, but they are not a panacea: They do not reduce inherent communication, and programs have significant miss rates from other sources as well. Latency usually grows with the size of the machine, since more nodes implies more communication relative to computation, more hops in the network for general communication, and likely more contention.

The goal of the protocols developed in the previous chapters has been to reduce the frequency of long-latency events and the bandwidth demands imposed on the communication media while providing a convenient programming model. The primary goal of hardware design has been to reduce the latency of data access while maintaining high, scalable bandwidth. Usually, we can improve bandwidth by throwing hardware at the problem, but latency is a more fundamental limitation. With careful design one can try not to exceed the inherent latency of the technology too much, but data access nonetheless takes time. So far, we have seen three ways to reduce the

latency of data access in a multiprocessor system, the first two the responsibility of the system and the third of the application.

- **Reduce the access time to each level of the storage hierarchy**. This requires careful attention to detail in making each step in the access path efficient. The processor-to-cache interface can be made very tight. The cache controller needs to act quickly on a miss to reduce the penalty in going to the next level. The network interface can be closely coupled with the node and designed to format, deliver, and handle network transactions quickly. The network itself can be designed to reduce routing delays, transfer time, and congestion delays. All of these efforts reduce the cost when access to a given level of the hierarchy is required. Nonetheless, the costs add up.

- **Reduce the likelihood that data accesses will incur high latency**: This is the basic job of automatic replication such as in caches, which take advantage of spatial and temporal locality in the program access pattern to keep the most important data close to the processor that accesses it. We can make replication more effective by tailoring the machine structure, for example, by providing a substantial amount of storage in each node.

- **Reduce the frequency of potential high-latency events in the application**. This involves decomposing and assigning computation to processors to reduce inherent communication, and structuring access patterns to increase spatial and temporal locality.

These system and application efforts to reduce latency help greatly, but often they do not suffice. There are also other potentially high-latency events than data access and communication, such as synchronization. This chapter discusses another approach to dealing with the latency that remains:

- **Tolerate the remaining latency**. That is, hide the latency from the processor's critical path by overlapping it with computation or other high-latency events. The processor is allowed to perform other useful work or even data access/communication while the high-latency event is in progress. The key to latency tolerance is in fact parallelism, since the overlapped activities must be independent of one another. The basic idea is very simple, and we use it all the time in our daily lives: If you are waiting for one thing, e.g. a load of clothes to finish in the washing machine, do something else, e.g. run an errand, while you wait (fortunately, data accesses do not get stolen if you don't keep an eye on them).

Latency tolerance cuts across all of the issues we have discussed in the book, and has implications for hardware as well as software, so it serves a useful role in 'putting it all together'. As we shall see, the success of latency tolerance techniques depends on both the characteristics of the application as well as on the efficiency of the mechanisms provided and the communication architecture.

Latency tolerance is already familiar to us from multiprogramming in uniprocessors. Recall what happens on a disk access, which is a truly long latency event. Since the access takes a long time, the processor does not stall waiting for it to complete. Rather, the operating system blocks the process that made the disk access and switches in another one, typically from another application, thus overlapping the latency of the access with useful work done on the other process. The blocked process is resumed later, hopefully after that access completes. This is latency tolerance: While the disk access itself does not complete any more quickly, and in this case the process that made the disk access sees its entire latency or even more since it may be blocked for even longer, the underlying resource (e.g. the processor) is not stalled and accomplishes other useful work in the meantime. While switching from one process to another via the operating system takes a lot

of instructions, the latency of disk access is high enough that this is very worthwhile. In this multiprogramming example, we do not succeed in reducing the execution time of any one process—in fact we might increase it—but we improve the system's throughput and utilization. More overall work gets done and more processes complete per unit time.

The latency tolerance that is our primary focus in this chapter is different from the above example in two important ways. First, we shall focus mostly on trying to overlap the latency with work from the *same* application; i.e. our goal will indeed be to use latency tolerance to reduce the execution time of a given application. Second, we are trying to tolerate not disk latencies but those of the memory and communication systems. These latencies are much smaller, and the events that cause them are usually not visible to the operating system. Thus, the time-consuming switching of applications or processes via the operating system is not a viable solution.

Before we go further, let us define a few terms that will be used throughout the chapter, including latency itself. The *latency* of a memory access or communication operation includes all components of the time from issue by the processor till completion. For communication, this includes the processor overhead, assist occupancy, transit latency, bandwidth-related costs, and contention. The latency may be for one-way transfer of communication or round-trip transfers, which will usually be clear from the context, and it may include the cost of protocol transactions like invalidations and acknowledgments in addition to the cost of data transfer. In addition to local memory latency and communication latency, there are two other types of latency for which a processor may stall. *Synchronization latency* is the time from when a processor issues a synchronization operation (e.g. lock or barrier) to the time that it gets past that operation; this includes accessing the synchronization variable as well as the time spent waiting for an event that it depends on to occur. Finally, *instruction latency* is the time from when an instruction is issued to the time that it completes in the processor pipeline, assuming no memory, communication or synchronization latency. Much of instruction latency is already hidden by pipelining, but some may remain due to long instructions (e.g. floating point divides) or bubbles in the pipeline. Different latency tolerance techniques are capable of tolerating some subset of these different types of latencies. Our primary focus will be on tolerating communication latencies, whether for explicit or implicit communication, but some of the techniques discussed are applicable to local memory, synchronization and instruction latencies, and hence to uniprocessors as well.

A communication from one node to another that is triggered by a single user operation is called a *message*, regardless of its size. For example, a `send` in the explicit message-passing abstraction constitutes a message, as does the transfer of a cache block triggered by a cache miss that is not satisfied locally in a shared address space (if the miss is satisfied locally its latency is called *memory latency*, if remotely then *communication latency*). Finally, an important aspect of communication for the applicability of latency tolerance techniques is whether it is initiated by the sender (source or producer) or receiver (destination or consumer) of the data. Communication is said to be *sender-initiated* if the operation that causes the data to be transferred is initiated by the process that has produced or currently holds the data, without solicitation from the receiver; for example, an unsolicited send operation in message passing. It is said to be *receiver-initiated* if the data transfer is caused or solicited by an operation issued by the process that needs the data, for example a read miss to nonlocal data in a shared address space. Sender- and receiver-initiated communication will be discussed in more detail later in the context of specific programming models.

Section 11.2 identifies the problems that result from memory and communication latency as seen from different perspectives, and provides an overview of the four major types of approaches to latency tolerance: block data transfer, precommunication, proceeding past an outstanding com-

munication event in the same thread, and multithreading or finding independent work to overlap in other threads of execution. These approaches are manifested as different specific techniques in different communication abstractions. This section also discusses the basic system and application requirements that apply to any latency tolerance technique, and the potential benefits and fundamental limitations of exploiting latency tolerance in real systems. Having covered the basic principles, the rest of the chapter examines how the four approaches are applied in the two major communication abstractions. Section 11.3 briefly discusses how they may be used in an explicit message-passing communication abstraction. This is followed by a detailed discussion of latency tolerance techniques in the context of a shared address space abstraction. The discussion for a shared address space is more detailed since latency is likely to be a more significant bottleneck when communication is performed through individual loads and stores than through flexibly sized transfers. Also, latency tolerance exposes interesting interactions with the architectural support already provided for a shared address space, and many of the techniques for hiding read and write latency in a shared address space are applicable to uniprocessors as well. Section 11.4 provides an overview of latency tolerance in this abstraction, and the next four sections each focus on one of the four approaches, describing the implementation requirements, the performance benefits, the tradeoffs and synergies among techniques, and the implications for hardware and software. One of the requirements across all techniques we will discuss is that caches be non-blocking or lockup-free, so Section 11.9discusses techniques for implementing lockup-free caches. Finally, Section 11.10 concludes the chapter.

11.2 Overview of Latency Tolerance

To begin our discussion of tolerating communication latency, let us look at a very simple example computation that will serve as an example throughout the chapter.

The example is a simple producer-consumer computation. A process P_A computes and writes the n elements of an array A, and another process P_B reads them. Each process performs some unrelated computation during the loop in which it writes or reads the data in A, and A is allocated in the local memory of the processor that P_A runs on. With no latency tolerance, the process generating the communication would simply issue a word of communication at a time—explicitly or implicitly as per the communication abstraction—and would wait till the communication message completes before doing anything else. This will be referred to as the baseline communication structure. The computation might look like that shown in Figure 11-1(a) with explicit message passing, and like that shown in Figure 11-1(b) with implicit, read-write communication in a shared address space.   In the former, it is the send operation issued by P_A that generates the actual communication of the data, while in the latter it is the read of *A[i]* by P_B.[1] We assume that the read stalls the processor until it completes, and that a synchronous send is used (as was described in Section 2.4.6). The resulting time-lines for the processes that initiate the communi-

---

1. The examples are in fact not exactly symmetric in their baseline communication structure, since in the message passing version the communication happens after each array entry is produced while in the shared address space version it happens after the entire array has been produced. However, implementing the fine-grained synchronization necessary for the shared address space version would require synchronization for each array entry, and the communication needed for this synchronization would complicate the discussion. The asymmetry does not affect the discussion of latency tolerance.
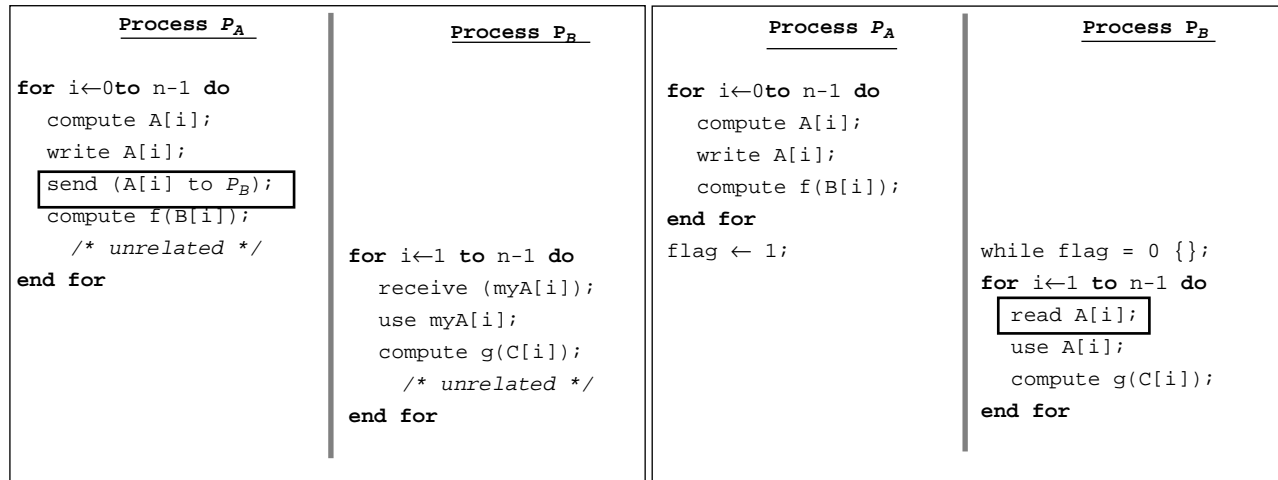
(a) Message Passing (b) Shared Address Space

Figure 11-1 Pseudocode for the example computation.

Pseudocode is shown for both explicit message passing and with implicit read-write communication in a shared address space, with no latency hiding in either case. The boxes highlight the operations that generate the data transfer.

cation are shown in Figure 11-2. A process spends most of its time stalled waiting for communication.

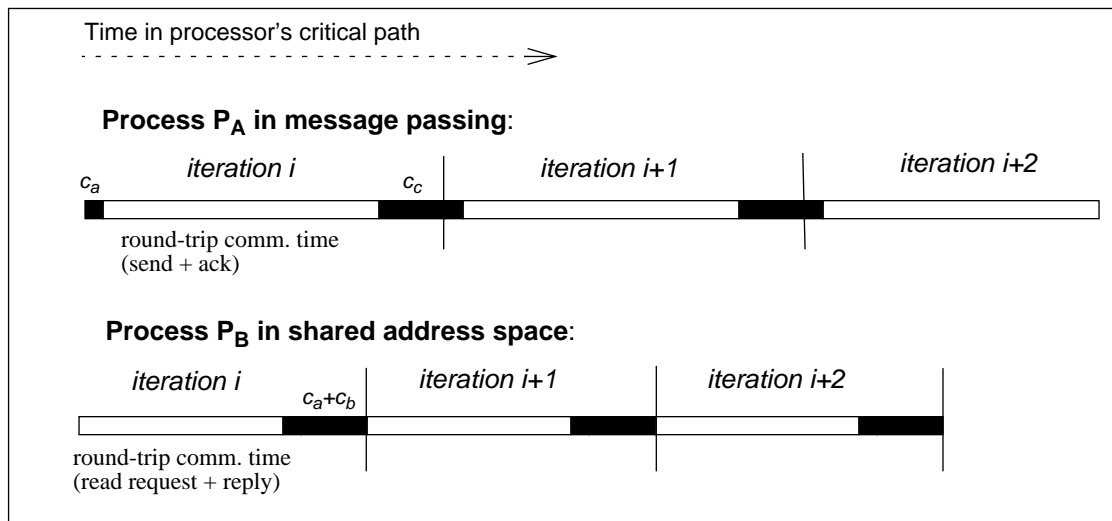Figure 11-2 Time-lines for the processes that initiate communication, with no latency hiding.

The black part of the time-line is local processing time (which in fact includes the time spent stalled to access local data), and the blank part is time spent stalled on communication. c_a, c_b, c_c are the times taken to compute an *A[i]* and to perform the unrelated computations *f(B[i])* and *g[C[i])*, respectively.

11.2.1 Latency Tolerance and the Communication Pipeline

Approaches to latency tolerance are best understood by looking at the resources in the machine and how they are utilized. From the viewpoint of a processor, the communication architecture from one node to another can be viewed as a pipeline. The stages of the pipeline clearly include the network interfaces at the source and destination, as well as the network links and switches along the way (see Figure 11-3). There may also be stages in the communication assist, the local
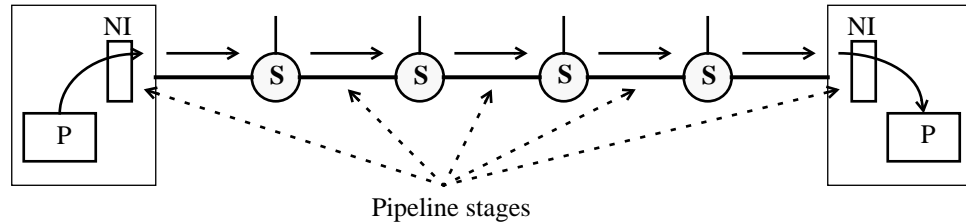
Pipeline stages

Figure 11-3 The network viewed as a pipeline for a communication sent from one processor to another.

The stages of the pipeline include the network interface (NI) and the hops between successive switches (S) on the route between the two processors (P).

memory/cache system, and even the main processor, depending on how the architecture manages communication. It is important to recall that the end-point overhead incurred on the processor itself cannot be hidden from the processor, though all the other components potentially can. Systems with high end-point overhead per message therefore have a difficult time tolerating latency. Unless otherwise mentioned, this overview will ignore the processor overhead; the focus at the end-points will be the assist occupancy incurred in processing, since this has a chance of being hidden too. We will also assume that this message initiation and reception cost is incurred as a fixed cost once per message (i.e. it is not proportional to message size).

Figure 11-4 shows the utilization problem in the baseline communication structure: Either the processor or the communication architecture is busy at a given time, and in the communication pipeline only one stage is busy at a time as the single word being transmitted makes its way from source to destination.[1] The goal in latency tolerance is therefore to overlap the use of these resources as much as possible. From a processor's perspective, there are three major types of overlap that can be exploited. The first is the communication pipeline between two nodes, which allows us to pipeline the transmission of multiple words through the network resources along the way, just like instruction pipelining overlaps the use of different resources (instruction fetch unit, register files, execute unit, etc.) in the processor. These words may be from the same message, if messages are larger than a single word, or from different messages. The second is the overlap in different portions of the network (including destination nodes) exploited by having communication messages outstanding with different nodes at once. In both cases, the communication of one word is overlapped with that of other words, so we call this overlapping communication with communication. The third major kind of overlap is that of computation with communication; i.e.

---

1. For simplicity, we ignore the fact that the width of the network link is often less than a word, so even a single word may occupy multiple stages of the network part of the pipeline.
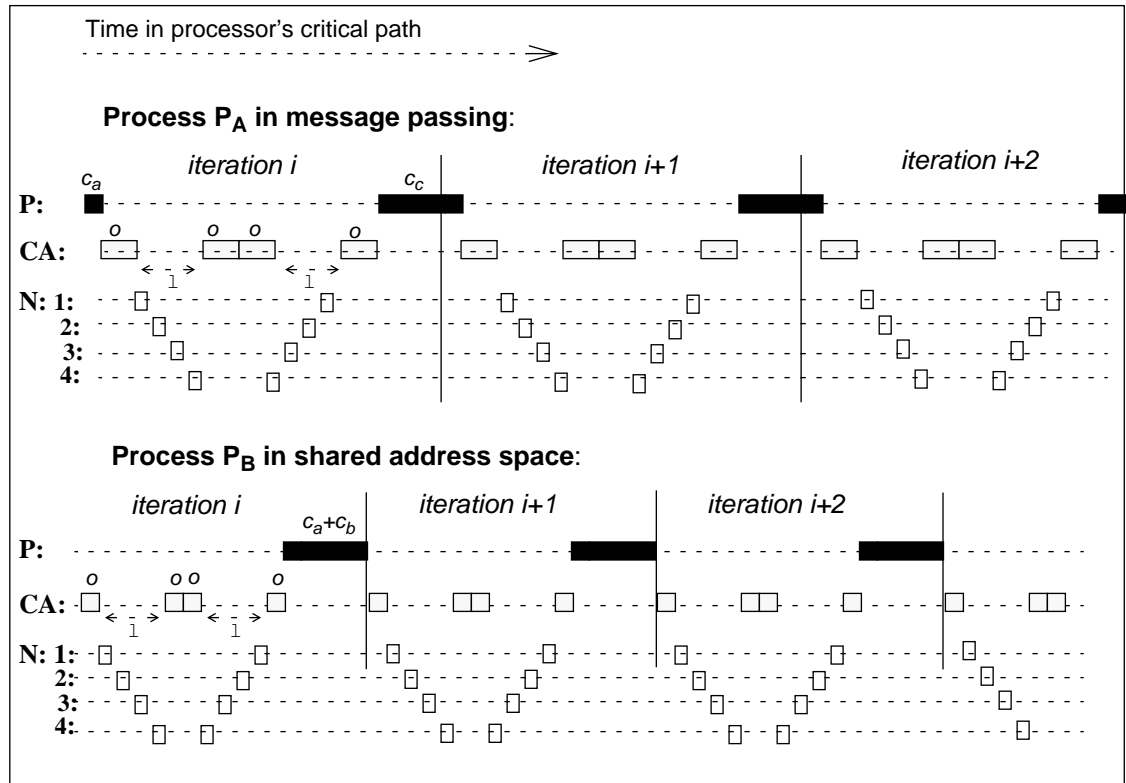
Figure 11-4 Time-lines for the message passing and shared address space programs with no latency hiding.

Time for a process is divided into that spent on the processor (P) including the local memory system, on the communication assist (CA) and in the network (N). The numbers under the network category indicate different hops or links in the network on the path of the message. The end-point overheads o in the message passing case are shown to be larger than in the shared address space, which we assume to be hardware-supported. l is the network latency, which is the time spent that does not involve the processing node itself.

a processor can continue to do useful local work while a communication operation it has initiated is in progress.

11.2.2 Approaches

There are four major approaches to actually exploiting the above types of overlap in hardware resources and thus tolerating latency. The first, which is called *block data transfer*, is to make individual messages larger so they communicate more than a word and can be pipelined through the network. The other three approaches are *precommunication*, *proceeding past communication in the same thread*, and *multithreading*. These are complementary to block data transfer, and their goal is to overlap the latency of a message, large or small, with computation or with other messages. Each of the four approaches can be used in both the shared address space and message passing abstractions, although the specific techniques and the support required are different in the two cases. Let us briefly familiarize ourselves with the four approaches.

Block data transfer

Making messages larger has several advantages. First, it allows us to exploit the communication pipeline between the two nodes, thus overlapping communication with communication (of different words). The processor sees the latency of the first word in the message, but subsequent words arrive every network cycle or so, limited only by the rate of the pipeline. Second, it serves the important purpose of amortizing the per-message overhead at the end-points over the large amount of data being sent, rather than incurring these overheads once per word (while message overhead may have a component that is proportional to the length of the message, there is invariably a large fixed cost as well). Third, depending on how packets are structured, this may also amortize the per-packet routing and header information. And finally, a large message requires only a single acknowledgment rather than one per word; this can mean a large reduction in latency if the processor waits for acknowledgments before proceeding past communication events, as well as a large reduction in traffic, end-point overhead and possible contention at the other end. Many of these advantages are similar to those obtained by using long cache blocks to transfer data from memory to cache in a uniprocessor, but on a different scale. Figure 11-5 shows the effect on the time-line of the example computation bunching up all the communication in a single large message in the explicit message passing case, while still using synchronous messages and without overlapping computation. For simplicity of illustration, the network stages have been collapsed into one (pipelined) stage.
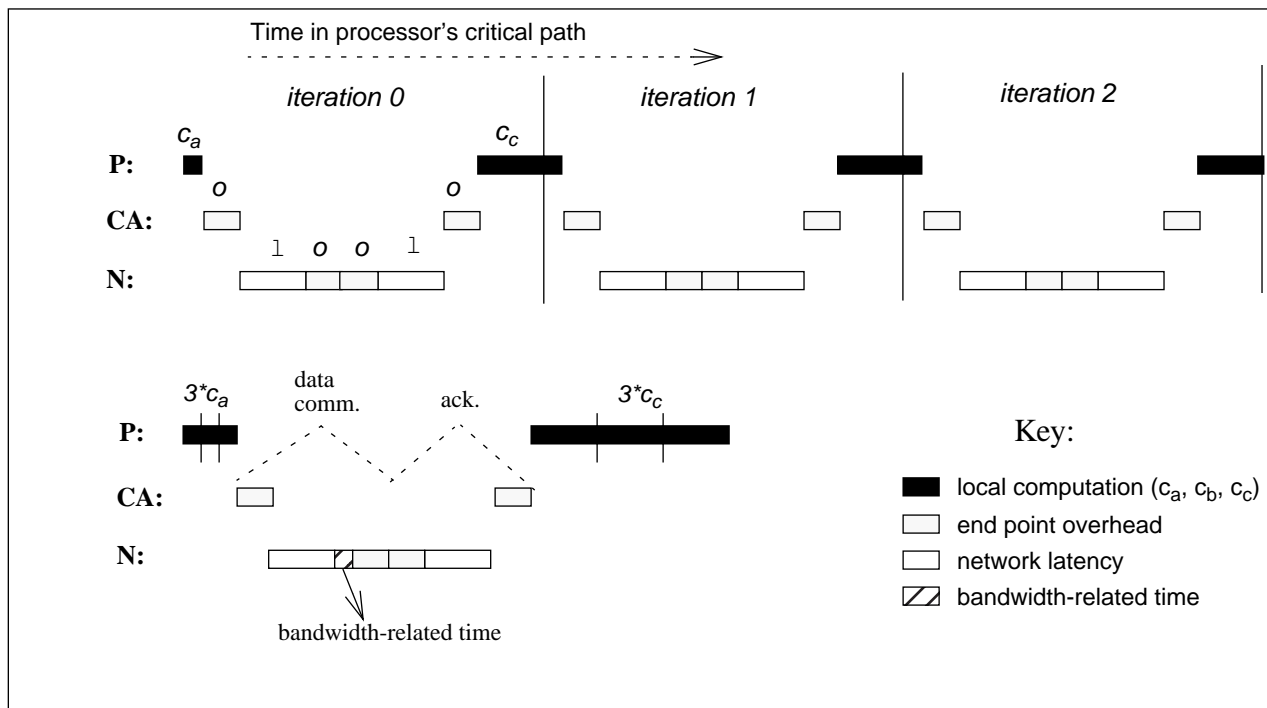
Figure 11-5 Effect of making messages larger on the time-line for the message passing program.

The figure assumes that three iterations of the loop are executed. The sender process (P_A) first computes the three values *A[0..2]* to be sent, then sends them in one message (the computation c_a), and then performs all three pieces of unrelated computation *f(B[0..2])*, i.e. the computation c_b. The overhead of the data communication is amortized, and only a single acknowledgment is needed. A bandwidth-related cost is added to the data communication, for the time taken to get the remaining words into the destination word after the first arrives, but this is tiny compared to the savings in latency and overhead. This is not required for the acknowledgment, which is small.

While making messages larger helps keep the pipeline between two nodes busy, it does not in itself keep the processor or communication assist busy while a message is in progress or keep other portion of the network busy. The other three approaches can address these problems as well. They are complementary to block data transfer, in that they are applicable whether messages are large or small. Let us examine them one by one.

Precommunication

This means generating the communication operation before the point where it naturally appears in the program, so that the operations are completed before data are actually needed[1]. This can be done either in software, by inserting a precommunication operation earlier in the code, or by having hardware detect the possibility of precommunication and generate it without an explicit software instruction. The operations that actually use the data typically remain where they are. Of course, the precommunication transaction itself should not cause the processor to stall until it completes. Many forms of precommunication require that long-latency events be predictable, so that hardware or software can anticipate them and issue them early.

Consider the interactions with sender- and receiver-initiated communication. Sender-initiated communication is naturally initiated soon after the sender produces the data. This has two implications. On one hand, the data may reach the receiver before actually needed, resulting in a form of precommunication for free for the receiver. On the other hand, it may be difficult for the sender to pull up the communication any earlier, making it difficult for the sender itself to hide the latency it sees through precommunication. Actual precommunication, by generating the communication operation earlier, is therefore more common in receiver-initiated communication. Here, communication is otherwise naturally initiated when the data are needed, which may be long after they have been produced.

Proceeding past communication in the same process/thread

The idea here is to let the communication operations be generated where they naturally occur in the program, but allow the processor to proceed past them and find other independent computation or communication that would come later in the same process to overlap with them. Thus, while precommunication causes the communication to be overlapped with other instructions from that thread that are *before* the point where the communication-generating operation appears in the original program, this technique causes it to be overlapped with instructions from *later* in the process. As we might imagine, this latency tolerance method is usually easier to use effectively with sender-initiated communication, since the work following the communication often does not depend on the communication. In receiver-initiated communication, a receiver naturally tries to access data only just before they are actually needed, so there is not much independent work to be found in between the communication and the use. It is, of course, possible to delay the actual use of the data by trying to push it further down in the instruction stream relative to other independent instructions, and compilers and processor hardware can exploit some overlap in this way. In either case, either hardware or software must check that the data transfer has completed before executing an instruction that depends on it.

---

1. Recall that several of the techniques, including this one, are applicable to hiding local memory access latency as well, even though we are speaking in terms of communication. We can think of that as communication with the memory system.

Multithreading

The idea here is similar to the previous case, except that the independent work is found not in the same process or thread but by switching to another thread that is mapped to run on the same processor (as discussed in Chapter 2, process and thread are used interchangeably in this book to mean a locus of execution with its own program counter and stack). This makes the latency of receiver-initiated communication easier to hide than the previous case and so this method lends itself easily to hiding latency from either a sender or a receiver. Multithreading implies that a given processor must have multiple threads that are concurrently executable or "ready" to have their next instruction executed, so it can switch from one thread to another when a long latency event is encountered. The multiple threads may be from the same parallel program, or from completely different programs as in our earlier multiprogramming example. A multithreaded program is usually no different from an ordinary parallel program; it is simply decomposed and assigned among P processors, where P is usually a multiple of the actual number of physical processors p, and P/p threads are mapped to the same physical processor.

11.2.3 Fundamental Requirements, Benefits and Limitations

Most of the chapter will focus on specific techniques and how they are exploited in the major programming models. We will pick this topic up again in Section 11.3. But first, it is useful to abstract away from specific techniques and understand the fundamental requirements, benefits and limitations of latency tolerance, *regardless* of the technique used. The basic analysis of benefits and limitations allows us to understand the extent of the performance improvements we can expect, and can be done based only on the goal of overlapping the use of resources and the occupancies of these resources.

Requirements

Tolerating latency has some fundamental requirements, regardless of approach or technique. These include extra parallelism, increased bandwidth, and in many cases more sophisticated hardware and protocols.

Extra parallelism, or *slackness*. Since the overlapped activities (computation or communication) must be independent of one another, the parallelism in the application must be greater than the number of processors used. The additional parallelism may be explicit in the form of additional threads, as in multithreading, but even in the other cases it must exist within a thread. Even two words being communicated in parallel from a process implies a lack of total serialization between them and hence extra parallelism.

Increased bandwidth. While tolerating latency hopefully reduces the execution time during which the communication is performed, it does not reduce the *amount* of communication performed. The same communication performed in less time means a higher rate of communication per unit time, and hence a larger bandwidth requirement imposed on the communication architecture. In fact, if the bandwidth requirements increase much beyond the bandwidth afforded by the machine, the resulting resource contention may slow down other unrelated transactions and may even cause latency tolerance to hurt performance rather than help it.

More sophisticated hardware and protocols. Especially if we want to exploit other forms of overlap than making messages larger, we have the following additional requirements of the hardware or the protocols used:

- the processor must be allowed to proceed past a long-latency operation before the operation completes; i.e. it should not stall upon a long-latency operation.
- a processor must be allowed to have multiple outstanding long-latency operations (e.g. send or read misses), if they are to be overlapped with one another.

These requirements imply that all latency tolerance techniques have significant costs. *We should therefore try to use algorithmic techniques to reduce the frequency of high latency events before relying on techniques to tolerate latency.* The fewer the long-latency events, the less aggressively we need to hide latency.

Potential Benefits

A simple analysis can give us good upper bounds on the performance benefits we might expect from latency tolerance, thus establishing realistic expectations. Let us focus on tolerating the latency of communication, and assume that the latency of local references is not hidden. Suppose a process has the following profile when running without any latency tolerance: It spends T_c cycles computing locally, T_{ov} cycles processing message overhead on the processor, T_{occ} (occupancy) cycles on the communication assist, and T_l cycles waiting for message transmission in the network. If we can assume that other resources can be perfectly overlapped with the activity on the main processor, then the potential speedup can be determined by a simple application of Amdahl's Law. The processor must be occupied for T_c+T_{ov} cycles; the maximum latency that we can hide from the processor is $T_l + T_{occ}$ (assuming $T_c+T_{ov} > T_l + T_{occ}$), so the maximum speedup

due to latency hiding is $\dfrac{T_c + T_{ov} + T_{occ} + T_l}{T_c + T_{ov}}$, or ($1 + \dfrac{T_{occ} + T_l}{T_c + T_o}$). This limit is an upper bound,

since it assumes perfect overlap of resources and that there is no extra cost imposed by latency tolerance. However, it gives us a useful perspective. For example, if the process originally spends at least as much time computing locally as stalled on the communication system, then the maximum speedup that can be obtained from tolerating communication latency is a factor of 2 even if there is no overhead incurred on the main processor. And if the original communication stall time is overlapped only with processor activity, not with other communication, then the maximum speedup is also a factor of 2 regardless of how much latency there was to hide. This is illustrated in Figure 11-6.

How much latency can actually be hidden depends on many factors involving the application and the architecture. Relevant application characteristics include the structure of the communication and how much other work is available to be overlapped with it. Architectural issues are the kind and costs of overlap allowed: how much of the end-point processing is performed on the main processor versus on the assist; can communication be overlapped with computation, other communication, or both; how many messages involving a given processor may be outstanding (in progress) at a time; to what extent can overhead processing can be overlapped with data transmission in the network for the same message; and what are the occupancies of the assist and the stages of the network pipeline.
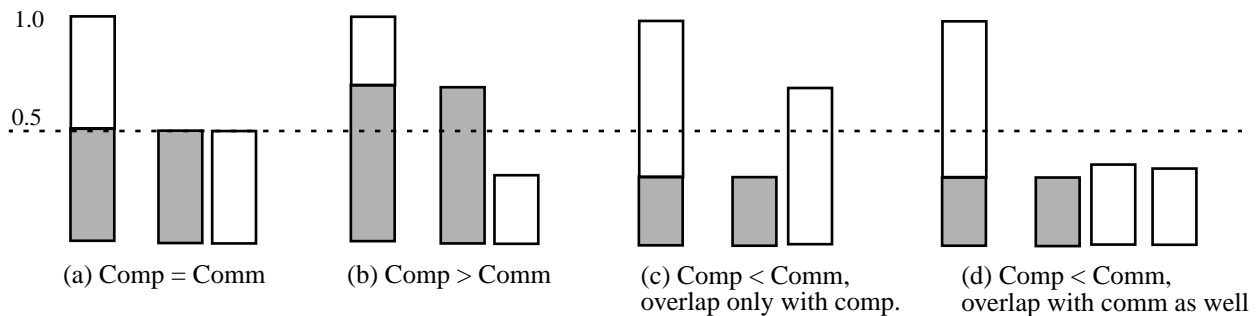
Figure 11-6 Bounds on the benefits from latency tolerance.

Each figure shows a different scenario. The gray portions represent computation time and the white portions communication time. The bar on the left is the time breakdown without latency hiding, while the bars on the right show the situation with latency hiding. When computation time (Comp.) equals communication time (Comm.), the upper bound on speedup is 2. When computation exceeds communication, the upper bound is less than 2, since we are limited by computation time (b). The same is true of the case where communication exceeds computation, but can be overlapped only with computation (c). The way to obtain a better speedup than a factor of two is to have communication time originally exceed computation time, but let communication be overlapped with communication as well (d).

Figure 11-7 illustrates the effects on the time-line for a few different kinds of message structuring and overlap. The figure is merely illustrative. For example, it assumes that overhead per message is quite high relative to transit latency, and does not consider contention anywhere. Under these assumptions, larger messages are often more attractive than many small overlapped messages for two reasons: first, they amortize overhead; and second, with small messages the pipeline rate might be limited by the end-point processing of each message rather than the network link speed. Other assumptions may lead to different results. Exercise b looks more quantitatively at an example of overlapping communication only with other communication.

Clearly, some components of communication latency are clearly easier to hide than others. For example, overhead incurred on the processor cannot be hidden from that processor, and latency incurred off-node—either in the network or at the other end—is generally easier to hide by overlapping with other messages than occupancy incurred on the assist or elsewhere within the node. Let us examine some of the key limitations that may prevent us from achieving the upper bounds on latency tolerance discussed above.

Limitations

The major limitations that keep the speedup from latency tolerance below the above optimistic limits can be divided into three classes: application limitations, limitations of the communication architecture, and processor limitations.

Application limitations: The amount of independent computation time that is available to overlap with the latency may be limited, so all the latency may not be hidden and the processor will have to stall for some of the time.Even if the program has enough work and extra parallelism, the structure of the program may make it difficult for the system or programmer to identify the concurrent operations and orchestrate the overlap, as we shall see when we discuss specific latency tolerance mechanisms.

Figure 11-7 Time-lines for different forms of latency tolerance. '

P indicates time spent on the main processor, 'A on the local communication assist, and 'N nonlocal (in network or on other nodes). Stipple patterns correspond to components of time as per legend in Figure 11-4.

Communication architecture limitations: Limitations of the communication architecture may be of two types: first, there may be restrictions on the number of messages or the number of words that can be outstanding from a n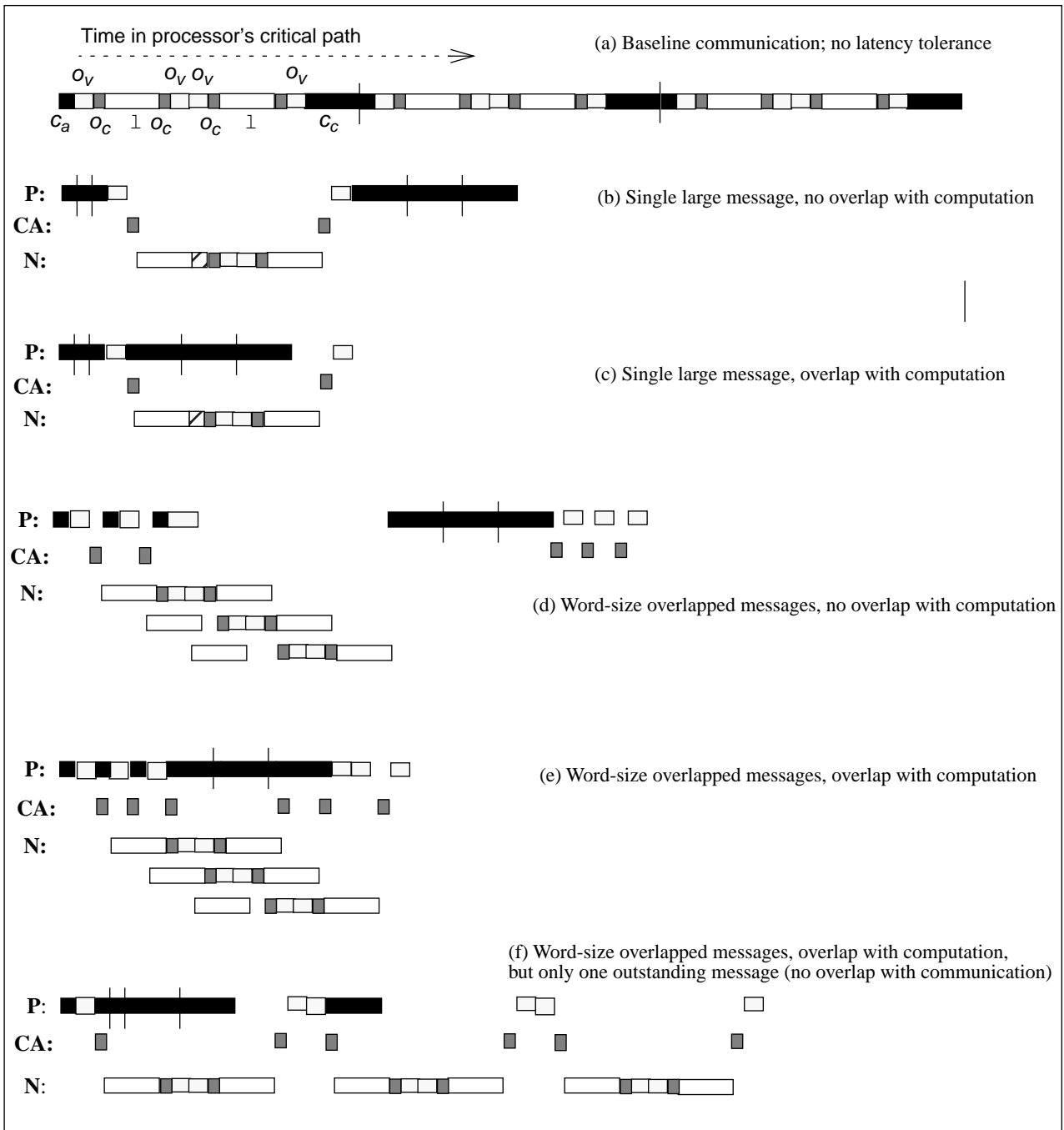ode at a time, and second, the performance parameters of the communication architecture may limit the latency that can be hidden.

Number of outstanding messages or words. With only one message outstanding, computation can be overlapped with both assist processing and network transmission. However, assist processing may or may not be overlapped with network transmission for that message, and the network pipeline itself can be kept busy only if the message is large. With multiple outstanding messages, all three of processor time, assist occupancy and network transmission can be overlapped even without any incoming messages. The network pipeline itself can be kept well utilized even when messages are very small, since several may be in progress simultaneously. Thus, for messages of a given size, more latency can be tolerated if each node allows multiple messages to be outstanding at a time. If L is the cost per message to be hidden, and r cycles of independent computation are available to be overlapped with each message, in the best case we need $\left\lceil \frac{L}{r} \right\rceil$ messages to be out-standing for maximal latency hiding. More outstanding messages do not help, since whatever latency could be hidden already has been. Similarly, the number or words outstanding is important. If k words can be outstanding from a processor at a time, from one or multiple messages, then in the best case the network transit time as seen by the processor can be reduced by almost a factor of k. More precisely, for one-way messages to the same destination it is reduced from $k*l$ cycles to $l + k/B$, where l is the network transit time for a bit and B is the bandwidth or rate of the communication pipeline (network and perhaps assist) in words per cycle.

Assuming that enough messages and words are allowed to be outstanding, the performance parameters of the communication architecture can become limitations as well. Let us examine some of these, assuming that the per-message latency of communication we want to hide is L cycles. For simplicity, we shall consider one-way data messages without acknowledgments.

Overhead. The message-processing overhead incurred on the processor cannot be hidden.

Assist occupancy: The occupancy of the assist can be hidden by overlapping with computation. Whether assist occupancy can be overlapped with other communication (in the same message or in different messages) depends on whether the assist is internally pipelined and whether assist processing can be overlapped with network transmission of the message data.

An end-point message processing time of o cycles per message (overhead or occupancy) establishes an upper bound on the frequency with which we can issue messages to the network (at best every o cycles). If we assume that on average a processor receives as many messages as it sends, and that the overhead of sending and receiving is the same, then the time spent in end-point processing per message sent is $2o$, so the largest number of messages that we can have outstanding from a processor in a period of L cycles is $\frac{L}{2o}$. If $2o$ is greater than the r cycles of computation we can overlap with each message, then we cannot have the $\frac{L}{r}$ messages outstanding that we would need to hide all the latency.

Point-to-point Bandwidth: Even if overhead is not an issue, the rate of initiating words into the network may be limited by the slowest link in the entire network pipeline from source to destina-

tion; i.e. the stages of the network itself, the node-to-network interface, or even the assist occupancy or processor overhead incurred per word. Just as an end-point overhead of o cycles between messages (or words) limits the number of outstanding messages (or words) in an L cycle period to $\frac{L}{o}$, a pipeline stage time of s cycles in the network limits the number of outstanding words to $\frac{L}{s}$.

Network Capacity: Finally, the number of messages a processor can have outstanding is also limited by the total bandwidth or data carrying capacity of the network, since different processors compete for this finite capacity. If each link is a word wide, a message of M words traveling h hops in the network may occupy up to $M*h$ links in the network at a given time. If each processor has k such messages outstanding, then each processor may require up to $M*h*k$ links at a given time. However, if there are D links in the network in all and all processors are transmitting in this way, then each processor on average can only occupy D/p links at a given time. Thus, the number of outstanding messages per processor k is limited by the relation $M \times h \times k < \frac{D}{p}$, or $k < \frac{D}{p \times M \times h}$.

Processor limitations: Spatial locality through long cache lines already allows more than one word of communication to be outstanding at a time. To hide latency beyond this in a shared address space with communication only through cache misses, a processor and its cache subsystem must allow multiple cache misses to be outstanding simultaneously. This is costly, as we shall see, so processor-cache systems have relatively small limits on this number of outstanding misses (and these include local misses that don't cause communication). Explicit communication has more flexibility in this regard, though the system may limit the number of messages that can be outstanding at a time.

It is clear from the above discussion that the communication architecture efficient (high bandwidth, low end point overhead or occupancy) is very important for tolerating latency effectively. We have seen the properties of some real machines in terms of network bandwidth, node-to-network bandwidth, and end-point overhead, as well as where the end-point overhead is incurred, in the last few chapters. From the data for overhead, communication latency, and gap we can compute two useful numbers in understanding the potential for latency hiding. The first is the ratio L/o for the limit imposed by communication performance on the number of messages that can be outstanding at a time in a period of L cycles, assuming that other processors are not sending messages to this one at the same time. The second is the inverse of the gap, which is the rate at which messages can be pipelined into the network. If L/o is not large enough to overlap the outstanding messages with computation, then the only way to hide latency beyond this is to make messages larger. The values of L/o for remote reads in a shared address space and for explicit message

passing using message passing libraries can be computed from Figures 7-31 and 7-32, and from the data in Chapter 8 as shown in Table 11-1.

Table 11-1 Number of messages outstanding at a time as limited by L/o ratio, and rate at which messages can be issued to the network. L here is the total cost of a message, including overhead. For a remote read, o is the overhead on the initiating processor, which cannot be hidden. For message passing, we assume symmetry in messages and therefore count o to be the sum of the send and receive overhead. (*) For the machines that have hardware support for a shared address space, there is very little time incurred on the main processor on a remote read, so most of the latency can be hidden if the processor and controller allow enough outstanding references. The gap here is determined by the occupancy of the communication assist.

Machine	Network Transaction		Remote Read		Machine	Network Transaction		Remote Read	
	L/2o	Msgs/ msec	L/o	Msgs/ msec		L/2o	Msgs/ msec	L/o	Msgs/ msec
TMC CM-5	2.12	250	1.75	161	*NOW-Ultra*	1,79	172	1.77	127
Intel Paragon	2.72	131	5.63	133	*Cray T3D*	1.00	345	1.13	2500*
Meiko CS-2	3.28	74	8.35	63.3	*SGI Origin*			*	5000*

For the message passing systems, it is clear that the number of messages that can be outstanding to overlap the cost of another message is quite small, due to the high per-message overhead. In the message-passing machines (e.g. nCube/2, CM-5, Paragon), hiding latency with small messages is clearly limited by the end-point processing overheads. Making messages larger is therefore likely to be more successful at hiding latency than trying to overlap multiple small messages. The hardware supported shared address space machines are limited much less by performance parameters of the communication architecture in their ability to hide the latency of small messages. Here, the major limitations tend to be the number of outstanding requests supported by the processor or cache system, and the fact that a given memory operation may generate several protocol transactions (messages), stressing assist occupancy.

Having understood the basics, we are now ready to look at individual techniques in the context of the two major communication abstractions. Since the treatment of latency tolerance for explicit message passing communication is simpler, the next section discusses all four general techniques in its context.

11.3 Latency Tolerance in Explicit Message Passing

To understand which latency tolerance techniques are most useful and how they might be employed, it is useful to first consider the structure of communication in an abstraction; in particular, how sender- and receiver-initiated communication are orchestrated, and whether messages are of fixed (small or large) or variable size.

11.3.1 Structure of Communication

The actual transfer of data in explicit message-passing is typically sender-initiated, using a send operation. Recall that a receive operation does not in itself cause data to be communicated, but rather copies data from an incoming buffer into the application address space. Receiver-initiated communication is performed by issuing a request message (itself a `send`) to the process that is

the source of the data. That process then sends the data back via another `send`.[1] The baseline example in Figure 11-1 uses synchronous sends and receives, with sender-initiated communication. A synchronous send operation has a communication latency equal to the time it takes to communicate all the data in the message to the destination, plus the time for receive processing, plus the time for an acknowledgment to be returned. The latency of a synchronous receive operation is its processing overhead, including copying the data into the application, plus the additional latency if the data have not yet arrived. We would like to hide these latencies, including overheads if possible, at both ends. Let us continue to assume that the overheads are incurred on a communication assist and not on the main processor, and see how the four classes of latency tolerance techniques might be applied in classical sender-initiated send-receive message passing.

11.3.2 Block Data Transfer

With the high end-point overheads of message passing, making messages large is important both for amortizing overhead and to enable a communication pipeline whose rate is not limited by the end-point overhead of message-processing. These two benefits can be obtained even with synchronous messages. However, if we also want to overlap the communication with computation or other messages, then we must use one or more of the techniques below (instead of or in addition to making messages larger). We have already seen how block transfer is implemented in message-passing machines in Chapter 7.

The other three classes of techniques, described below, try to overlap messages with computation or other messages, and must therefore use asynchronous messages so the processor does not stall waiting for each message to complete before proceeding. While they can be used with either small or large (block transfer) messages, and are in this sense complementary to block transfer, we shall illustrate them with the small messages that are in our baseline communication structure.

11.3.3 Precommunication

In our example of Figure 11-1, we can actually pull the sends up earlier than they actually occur, as shown in Figure 11-8. We would also like to pull the receives on P_B up early enough before the data are needed by P_B, so that we can overlap the receive overhead incurred on the assist with other work on the main processor. Figure 11-8 shows a possible precommunicating version of the message passing program. The loop on P_A is split in two. All the sends are pulled up and combined in one loop, and the computations of the function *f(B[])* are postponed to a separate loop. The sends are asynchronous, so the processor does not stall at them. Here, as in all cases, the use of asynchronous communication operations means that probes must be used as appropriate to ensure the data have been transferred or received.

Is this a good idea? The advantage is that messages are sent to the receiver as early as possible; the disadvantages are that no useful work is done on the sender between messages, and the pres-

---

1. In other abstractions built on message passing machines and discussed inChapter 7, like remote procedure call or active messages, the receiver sends a request message, and a handler that processes the request at the data source sends the data back without involving the application process there. However, we shall focus on classical send-receive message passing which dominates at the programming model layer.

sure on the buffers at the receiver is higher since messages may arrive much before they are needed. Whether the net result is beneficial depends on the overhead incurred per message, the number of messages that a process is allowed to have outstanding at a time, and how much later than the sends the receives occur anyway. For example, if only one or two messages may be outstanding, then we are better off interspersing computation (of *f(B[])*) between asynchronous sends, thus building a pipeline of communication and computation in software instead of waiting for those two messages to complete before doing anything else. The same is true if the overhead incurred on the assist is high, so we can keep the processor busy while the assist is incurring this high per-message overhead. The ideal case would be if we could build a balanced software pipeline in which we pull sends up but do just the right amount of computation between message sends.

```
            Process P_A                              Process P_B

  for i←0 to n-1 do                       a_receive (myA[0] from P_A);
     compute A[i];                        for i←0 to n-2 do
     write A[i];                            a_receive (myA[i+1] from P_A);
     a_send (A[i] to proc.P_B);            while (!recv_probe(myA[i]) {};
  end for                                   use myA[i];
  for i←0 to n-1 do                         compute g(C[i]);
     compute f(B[i]);                     end for
                                         while (!received(myA[n-1]) {};
                                         use myA[n-1];
                                         compute g(C[n-1])
```
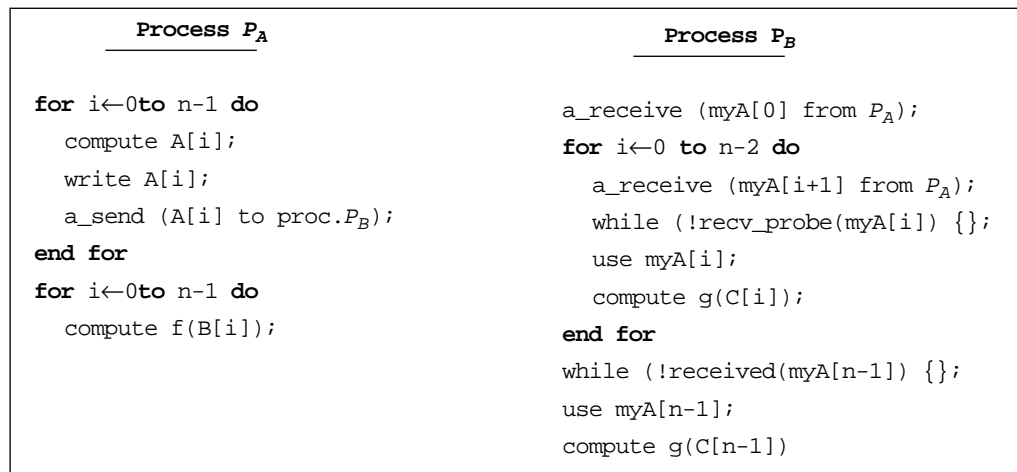
Figure 11-8 Hiding latency through precommunication in the message-passing example.

In the pseudocode, a_send and a_receive are asynchronous send and receive operations, respectively.

Now consider the receiver P_B. To hide the latency of the receives through precommunication, we try to pull them up earlier in the code and we use asynchronous receives. The a_receive call simply posts the specification of the receive in a place where the assist can see it, and allows the processor to proceed. When the data come in, the assist is notified and it moves them to the application data structures. The application must check that the data have arrived (using the recv_probe() call) before it can use them reliably. If the data have already arrived when the a_receive is posted (pre-issued), then what we can hope to hide by pre-issuing is the overhead of the receive processing; otherwise, we might hide both transit time and receive overhead.

Receive overhead is usually larger than send overhead, as we saw in Chapter 7, so if many asynchronous receives are to be issued (or pre-issued) it is usually beneficial to intersperse computation between them rather than issue them back to back. One way to do this is to build a *software pipeline* of communication and computation, as shown in Figure 11-8. Each iteration of the loop issues an a_receive for the data needed for the *next* iteration, and then does the processing for the current iteration (using a recv_probe to ensure that the message for the current iteration has arrived before using it). The hope is that by the time the next iteration is reached, the message for

which the `a_receive` was posted in the current iteration will have arrived and incurred its overhead on the assist, and the data will be ready to use. If the receive overhead is high relative to the computation per iteration, the `a_receive` can be several iterations ahead instead of just one issued.

The software pipeline has three parts. What was described above is the *steady-state* loop. In addition to this loop, some work must be done to start the pipeline and some to wind it down. In this example, the prologue must post a receive for the data needed in the first iteration of the steady-state loop, and the epilogue must process the code for the last iteration of the original loop (this last interaction is left out of the steady-state loop since we do not want to post an asynchronous receive for a nonexistent next iteration). A similar software pipeline strategy could be used if communication were truly receiver-initiated, for example if there were no send but the `a_receive` sent a request to the node running P_A and an handler there reacted to the request by supplying the data. In that case, the precommunication strategy is known as *prefetching* the data, since the receive (or get) truly causes the data to be fetched across the network. We shall see prefetching in more detail in the context of a shared address space in Section 11.7, since there the communication is very often truly receiver-initiated.

11.3.4 Proceeding Past Communication and Multithreading

Now suppose we do not want to pull communication up in the code but leave it where it is. One way to hide latency in this case is to simply make the communication messages asynchronous, and proceed past them to either computation or other asynchronous communication messages. We can continue doing this until we come to a point where we depend on a communication message completing or run into a limit on the number of messages we may have outstanding at a time. Of course, as with prefetching the use of asynchronous receives means that we must add probes or synchronization to ensure that data have reached their destinations before we try to use them (or on the sender side that the application data we were sending has been copied or sent out before we reuse the corresponding storage).

To exploit multithreading, as soon as a process issues a send or receive operation, it suspends itself and another ready-to-run process or thread from the application is scheduled to run on the processor. If this thread issues a send or receive, then it too is suspended and another thread switched in. The hope is that by the time we run out of threads and the first thread is rescheduled, the communication operation that thread issued will have completed. The switching and management of threads can be managed by the message passing library using calls to the operating system to change the program counter and perform other protected thread-management functions. For example, the implementation of the send primitive might automatically cause a thread switch after initiating the send (of course, if there is no other ready thread on that processing node, then the same thread will be switched back in).

Switching a thread requires that we save the processor state needed to restart it that will otherwise be wiped out by the new thread. This includes the processor registers, the program counter, the stack pointer, and various processor status words. The state must be restored when the thread is switched back in. Saving and restoring state in software is expensive, and can undermine the benefits obtained from multithreading. Some message-passing architectures therefore provide hardware support for multithreading, for example by providing multiple sets of registers and program counters in hardware. Note that architectures that provide asynchronous message handlers that are separate from the application process but run on the main processor are essentially multi-

threaded between the application process and the handler. This form of multithreading between applications and message handlers can be supported in software, though some research architectures also provide hardware support to make it more efficient (e.g. the message-driven processor of the J-Machine [DFK+92,NWD93]). We shall discuss these issues of hardware support in more detail when we discuss multithreading in a shared address space in Section 11.8.

11.4 Latency Tolerance in a Shared Address Space

The rest of this chapter focuses on latency tolerance in the context of hardware-supported shared address space. This discussion will be a lot more detailed for several reasons. First, the existing hardware support for communication brings the techniques and requirements much closer to the architecture and the hardware-software interface. Second, the implicit nature of long-latency events (such as communication) makes it almost necessary that much of the latency tolerance be orchestrated by the system rather than the user program. Third, the granularity of communication and efficiency of the underlying fine-grained communication mechanisms require that the techniques be hardware-supported if they are to be effective. And finally, since much of the latency being hidden is that of reads (*read latency*), writes (*write latency*) and perhaps instructions—not explicit communication operations like sends and receives—most of the techniques are applicable to the wide market of uniprocessors as well. In fact, since we don't know ahead of time which read and write operations will generate communication, latency hiding is treated in much the same way for local accesses and communication in shared address space multiprocessors. The difference is in the magnitude of the latencies to be hidden, and potentially in the interactions with a cache coherence protocol.

As with message passing, let us begin by briefly examining the structure of communication in this abstraction before we look at the actual techniques. Much of our discussion will apply to both cache-coherent systems as well as those that do not cache shared data, but some of it, including the experimental results presented, will be specific to cache-coherent systems. For the most part, we assume that the shared address space (and cache coherence) is supported in hardware, and the default communication and coherence are at the small granularity of individual words or cache blocks. The experimental results presented in the following sections will be taken from the literature. These results use several of the same applications that we have used earlier in this book, but they typically use different versions with different communication to computation ratios and other behavioral characteristics, and they do not always follow the methodology outlined in Chapter 4. The system parameters used also vary, so the results are presented purely for illustrative purposes, and cannot be compared even among different latency hiding techniques in this chapter.

11.4.1 Structure of Communication

The baseline communication is through reads and writes in a shared address space, and is called *read-write communication* for convenience. Receiver-initiated communication is performed with read operations that result in data from another processor's memory or cache being accessed. This is a natural extension of data access in the uniprocessor program model: Obtain words of memory when you need to use them.

If there is no caching of shared data, sender-initiated communication may be performed through writes to data that are allocated in remote memories.[1] With cache coherence, the effect of writes is more complex: Whether writes lead to sender or receiver initiated communication depends on the cache coherence protocol. For example, suppose processor P_A writes a word that is allocated in P_B's memory. In the most common case of an invalidation-based cache coherence protocol with writeback caches, the write will only generate a read exclusive or upgrade request and perhaps some invalidations, and it may bring data to itself if it does not already have the valid block in its cache, but it will not actually cause the newly written data to be transferred to P_B. While requests and invalidations constitute communication too, the actual communication of the new value from P_A to P_B will be generated by the later read of the data by P_B. In this sense, it is receiver-initiated. Or the data transfer may be generated by an asynchronous replacement of the data from P_A's cache, which will cause it to go back to its "home" in P_B's memory. In an update protocol, on the other hand, the write itself will communicate the data from P_A to P_B if P_B had a cached copy.

Whether receiver-initiated or sender-initiated, the communication in a hardware-supported read-write shared address space is naturally fine-grained, which makes tolerance latency particularly important. The different approaches to latency tolerance are better suited to different types and magnitudes of latency, and have achieved different levels of acceptance in commercial products. Other than block data transfer, the other three approaches that try to overlap load and store accesses with computation or other accesses require support in the microprocessor architecture itself. Let us examine these approaches in some detail in the next four sections.

11.5 Block Data Transfer in a Shared Address Space

In a shared address space, the coalescing of data and the initiation of block transfers can be done either explicitly in the user program or transparently by the system, either by hardware or software. For example, the prevalent use of long cache blocks on modern systems can itself be seen as a means of transparent small block transfers in cache-block sized messages, used effectively when a program has good spatial locality on communicated data. Relaxed memory consistency models further allow us to buffer words or cache blocks and send them in coalesced messages only at synchronization points, a fact utilized particularly by software shared address space systems as seen in Chapter 9. However, let us focus here on explicit initiation of block transfers.

11.5.1 Techniques and Mechanisms

Explicit block transfers are initiated by issuing a command similar to a `send` in the user program, as shown for the simple example in Figure 11-9. The send command is interpreted by the communication assist, which transfers the data in a pipelined manner from the source node to the destination. At the destination, the communication assist pulls the data words in from the network interface and stores them in the specified locations. The path is shown in Figure 11-10. There are two major differences from send-receive message passing, both of which arise from the fact that

---

1. An interesting case is the one in which a processor writes a word that is allocated in a different processor's memory, and a third processor reads the word. In this case, we have two communication events, one "sender-initiated" and one "receiver-initiated".
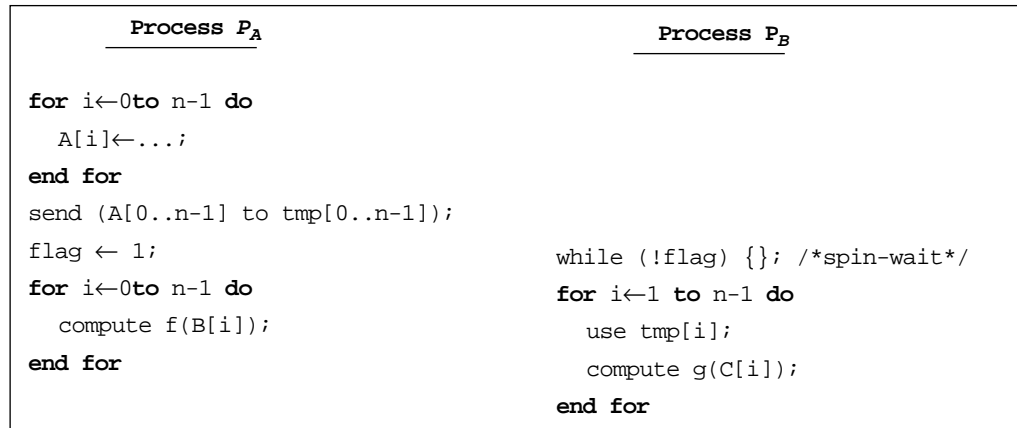
```
              Process P_A                              Process P_B

for i←0 to n-1 do
  A[i]←...;
end for
send (A[0..n-1] to tmp[0..n-1]);
flag ← 1;                              while (!flag) {}; /*spin-wait*/
for i←0 to n-1 do                      for i←1 to n-1 do
  compute f(B[i]);                        use tmp[i];
end for                                   compute g(C[i]);
                                       end for
```

Figure 11-9 Using block transfer in a shared address space for the example of Figure 11-1.

The array *A* is allocated in processor P_A's local memory, and *tmp* in processor P_B's local memory.

the sending process can directly specify the program data structures (virtual addresses) where the data are to be placed at the destination, since these locations are in the shared address space. On one hand, this removes the need for receive operations in the programming model, since the incoming message specifies where the data should be put in the program address space. It may also remove the need for system buffering in main memory at the destination, if the destination assist is available to put the data directly from the network interface into the program data structures in memory. On the other hand, another form of synchronization (like spinning on a flag or blocking) must be used to ensure that data arrive before they are used by the destination process, but not before they are welcome. It is also possible to use receiver-initiated block transfer, in which case the request for the transfer is issued by the receiver and transmitted to the sender's commnication assist.

The communication assist on a node that performs the block transfer could be the same one that processes coherence protocol transactions, or a separate DMA engine dedicated to block transfer. The assist or DMA engine can be designed with varying degrees of aggressiveness in its functionality; for example, it may allow or disallow transfers of data that are not contiguous in memory at the source or destination ends, such as transfers with uniform stride or more general scatter-gather operations discussed in Chapter 2. The block transfer may leverage off the support already provided for efficient transfer of cache blocks, or it may be a completely separate mechanism. Particularly since block transfers may need to interact with the coherence protocol in a coherent shared address space, we shall assume that the block transfer uses pipelined transfers of cache blocks; i.e. the block transfer engine is a part of the source that reads cache blocks out of memory and transfers them into the network. Figure 11-10 shows the steps in a possible implementation of block transfer in a cache-coherent shared address space.

11.5.2 Policy Issues and Tradeoffs

Two policy issues are of particular interest in block transfer: those that arise from interactions with the underlying shared address space and cache coherence protocol, and the question of where the block transferred data should be placed in the destination node.
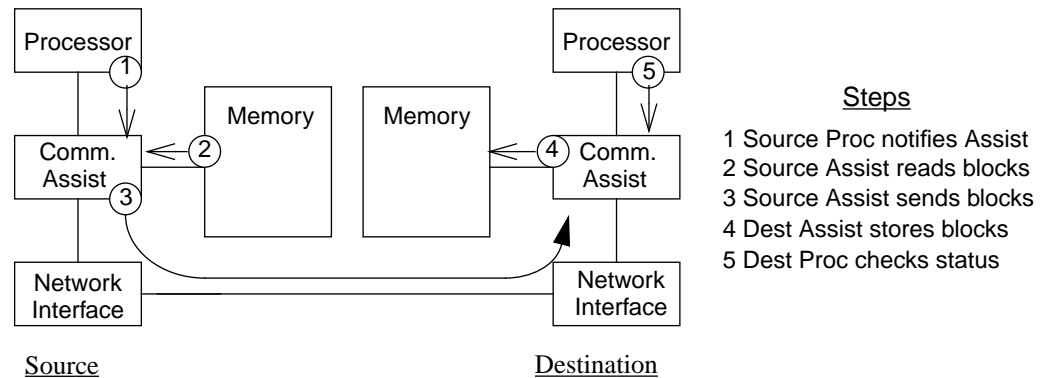
Figure 11-10 Path of a block transfer from a source node to a destination node in a shared address space machine.

The transfer is done in terms of cache blocks, since that is the granularity of communication for which the machine is optimized.

Interactions with the Cache Coherent Shared Address Space

The first interesting interaction is with the shared address space itself, regardless of whether it includes automatic coherent caching or not. The shared data that a processor "sends" in a block transfer may not be allocated in its local memory, but in another processing node's memory or even scattered among other node's memories. The main policy choices here are to (a) disallow such transfers, (b) have the initiating node retrieve the data from the other memories and forward them one cache block at a time in a pipelined manner, or (c) have the initiating node send messages to the owning nodes' assists asking them to initiate and handle the relevant transfers.

The second interaction is specific to a coherent shared address space, since now the same data structures may be communicated using two different protocols: block transfer and cache coherence. Regardless of which main memory the data are allocated in, they may be cached in (i) the sender's cache in dirty state, (ii) the receiver's cache in shared state, or (iii) another processor's cache (including the receiver's) in dirty state. The first two cases create what is called the *local coherence* problem, i.e. ensuring that the data sent are the latest values for those variables on the sending node, and that copies of those data on the receiving node are left in coherent state after the transfer. The third case is called the *global coherence* problem, i.e ensuring that the values transferred are the latest values for those variables anywhere in the system (according to the consistency model), and that the data involved in the transfer are left in coherent state in the whole system. Once again, the choices are to provide no guarantees in any of these three cases, to provide only local but not global coherence, or to provide full global coherence. Each successive step makes the programmer's job easier, but imposes more requirements on the communication assist. This is because the assist is now responsible for checking the state of every block being transferred, retrieving data from the appropriate caches, and invalidating data in the appropriate caches. The data being transferred may not be properly aligned with cache blocks, which makes interactions with the block-level coherence protocol even more complicated, as does the fact that the directory information for the blocks being transferred may not be present at the sending node More details on implementing block transfer in cache coherent machines can be found in the literature [KuA93,HGD+94,HBG+97]. Programs that use explicit message passing are simpler in

this regard: they require local coherence but not global coherence, since any data that they can access—and hence transfer—are allocated in their private address space and cannot be cached by any other processors.

For block transfer to maintain even local coherence, the assist has some work to do for every cache block and becomes an integral part of the transfer pipeline. The occupancy of the assist can therefore easily become the bottleneck in transfer bandwidth, affecting even those blocks for which no interaction with caches is required. Global coherence may require the assist to send network transactions around before sending each block, reducing bandwidth dramatically. It is therefore important that a block transfer system have a "pay for use" property; i.e. providing coherent transfers should not significantly hurt the performance of transfers that do not need coherence, particularly if the latter are a common case. For example, if block transfer is used in the system only to support explicit message passing programs, and not to accelerate coarse-grained communication in an otherwise read-write shared address space program, then it may make sense to provide only local coherence but not global coherence. But in the former case, global coherence may in fact be important if we want to provide true integration of block transfer with a coherent read-write shared address space, and not make the programming model restrictive. As a simple example of why global coherence may be needed, consider false sharing. The data that processor P1 wants to send P2 may be allocated in P1's local memory and produced by P1, but another processor P3 may have written other words on those cache blocks more recently. The cache blocks will therefore be in invalid state in P1's cache, and the latest copies must be retrieved from P3 in order to be sent to P2. False sharing is only an example: It is not difficult to see that in a true shared address space program the words that P1 sends to P2 might themselves have been written most recently by another processor.

Where to Place the Transferred Data?

The other interesting policy issue is whether the data that are block transferred should be placed in the main memory of the destination node, in the cache, or in both. Since the destination node will read the data, having the data come into the cache is useful. However, it has some disadvantages. First, it requires intervening in the processor cache, which is expensive on modern systems and may prevent the processor from accessing the cache at the same time. Second, unless the block transferred data are used soon they may be replaced before they are used, so we should transfer data into main memory as well. And third and most dangerous, the transferred data might replace other useful data from the cache, perhaps even data that are currently in the processor's active working set. For these reasons, large block transfer into a small first-level cache are likely to not be a good idea, while transfers into a large or second-level cache may be more useful.

11.5.3 Performance Benefits

In a shared address space, there are some new advantages for using large explicit transfer compared to communicating implicitly in cache block sized messages. However, some of the advantages discussed earlier are compromised and there are disadvantages as well. Let us first discuss the advantages and disadvantages qualitatively, and then look at some performance data on real programs from the literature and some analytical observations.

Potential Advantages and Disadvantages

The major sources of performance advantage are outlined below. The first two were discussed in the context of message passing, so we only point out the differences here.

- *Amortized per-message overhead.* This advantage may be less important in a hardware-supported shared address space, since the end-point overhead is already quite small. In fact, explicit, flexibly sized transfers have higher end-point overhead than small fixed-size transfers, as discussed in Chapter 7, so the per-communication overhead is likely to be substantially larger for block transfer than for read-write communication. This turns out to be a major stumbling block for block transfer on modern cache-coherent machines, which currently use the facility more for operating system operations like page migration than for application activity. However, in less efficient communication architectures, such as those that use commodity parts, the overhead per message can be large even for cache block communication and the amortization due to block transfer can be very important.

- *Pipelined transfer* of large chunks of data.

- *Less wasted bandwidth*: In general, the larger the message, the less the relative overhead of headers and routing information, and the higher the *payload* (the fraction of data transferred that is useful to the end application). This advantage may be lost when the block transfer is built on top of the existing cache line transfer mechanism, in which case again the header information may be nonetheless sent once per cache block. When used properly, block transfer can also reduce the number of protocol messages (e.g. invalidations, acknowledgments) that may have otherwise been sent around in an invalidation-based protocol.

- *Replication of transferred data in main memory* at the destination, since the block transfer is usually done into main memory. This reduces the number of capacity misses at the destination that have to be satisfied remotely, as in COMA machines. However, without a COMA architecture it implies that the user must manage the coherence and replacement of the data that have been block transferred and hence replicated in main memory.

- *Bundling of synchronization with data transfer.* Synchronization notifications can be piggybacked on the same message that transfers data, rather than having to send separate messages for data and synchronization. This reduces the number of messages needed, though the absence of an explicit receive operation implies that functionally synchronization has still to be managed separately from data transfer (i.e. there is no send-receive match that provides synchronization) just as in asynchronous send-receive message passing.

The sources of potential performance disadvantage are:

- *Higher overhead* per transfer, as discussed in the first point above.

- *Increased contention*. Long messages tend to incur higher contention, both at the end points and in the network. This is because longer messages tend to occupy more resources in the network at a time, and because the latency tolerance they provide places greater bandwidth demands on the communication architecture.

- *Extra work*. Programs may have to do extra work to organize themselves so communication can be performed in large transfers (if this can be done effectively at all). This extra work may turn out to have a higher cost than the benefits achieved from block transfer (see Section 3.7 in Chapter 3).

The following simple example illustrate the performance improvements that can be obtained by using block transfers rather than reads and writes, particularly from reduced overhead and pipelined data transfer.

Example 11-1

Suppose we want to communicate 4KB of data from a source node to a destination node in a cache-coherent shared address space machine with a cache block size of 64 bytes. Assume that the data are contiguous in memory, so spatial locality is exploited perfectly (Exercise 11.4 discusses the issues that arise when spatial locality is not good). It takes the source assist 40 processor cycles to read a cache block out of memory, 50 cycles to push a cache block out through the network interface, and the same numbers of cycles for the complementary operations at the receiver. Assume that the local read miss latency is 60 cycles, the remote read miss latency 180 cycles, and the time to start up a block transfer 200 cycles. What is the potential performance advantage of using block transfer rather than communication through cache misses, assuming that the processor blocks on memory operations until they complete?

Answer

The cost of getting the data through read misses, as seen by the processor, is 180*(4096/64) = 11520 cycles. With block transfer, the rate of the transfer pipeline is limited by max(40,50,50) or 50 cycles per block. This brings the data to the local memory at the destination, from where they will be read by the processor through local misses, each of which costs 60 cycles. Thus, the cost of getting the data to the destination processor with block transfer is 200 + (4096/64)*(50+60) or 7240 cycles. Using block transfer therefore gives us a speedup of 11520/7240 or 1.6.

Another way to achieve block transfers is with vector operations, e.g. a vector read from remote memory. In this case, a single instruction causes the data to appear in the (vector) registers, so individual loads and stores are not required even locally and there is a savings in instruction bandwidth. Vector registers are typically managed by software, which has the disadvantages of aliasing and tying up register names but the advantage of not suffering from cache conflicts. However, many high-performance systems today do not include vector operations, and we shall focus on block transfers that still need individual load and store operations to access and use the data.

Performance Benefits and Limitations in Real Programs

Whether block transfer can be used effectively in real programs depends on both program and system characteristics. Consider a simple example amenable to block transfer, a near-neighbor grid computation, to see how the performance issues play out. Let us ignore the complications discussed above by assuming that the transfers are done from the source main memory to the destination main memory and that the data are not cached anywhere. We assume a four-dimensional array representation of the two-dimensional grid, and that a processor's partition of the grid is allocated in its local memory.

Instead of communicating elements at partition boundaries through individual reads and writes, a process can simply send a single message containing all the appropriate boundary elements to its neighbor, as shown in Figure 11-11. The communication to computation ratio is proportional to
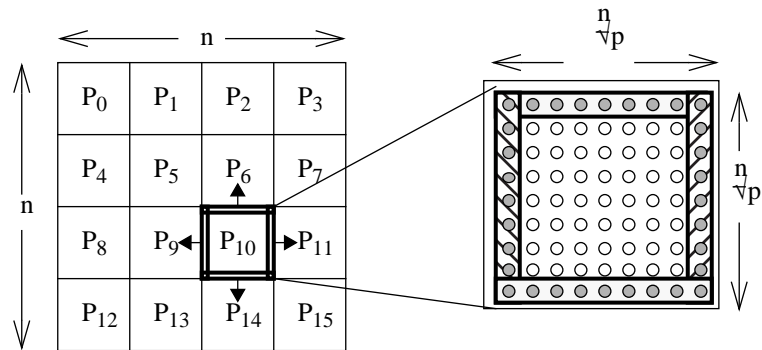
Figure 11-11   The use of block transfer in a near-neighbor grid program.

A process can send and entire boundary subrow or subcolumn of its partition to the process that owns the neighboring partition in a single large message. The size of each message is $\frac{n}{\sqrt{p}}$ elements.

$\frac{\sqrt{p}}{n}$. Since block transfer is intended to improve communication performance, this in itself would tend to increase the relative effectiveness of block transfer as the number of processors increases. However, the size of an individual block transfer is $\frac{n}{\sqrt{p}}$ elements and therefore decreases with p. Smaller transfers make the additional overhead of initiating a block transfer relatively more significant, reducing the effectiveness of block transfer. These two factors combine to yield a sweet-spot in the number of processors for which block transfer is most effective for a given grid size, yielding a performance improvement curve for block transfer that might look like that in Figure 11-12. The sweet-spot moves to larger numbers of processors as the grid size increases.
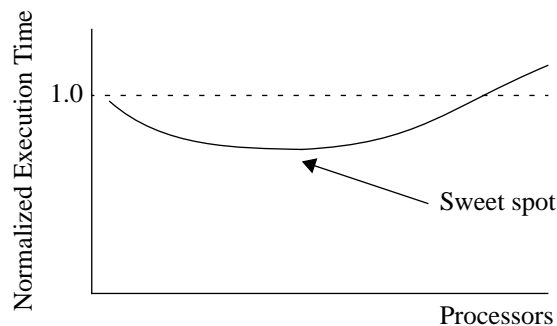
Figure 11-12   Relative performance improvement with block transfer.

The figure shows the execution time using block transfer normalized to that using loads and stores. The sweet spot occurs when communication is high enough that block transfer matters, and the transfers are also large enough that overhead doesn't dominate.

Figure 11-13 illustrates this effect for a real program, namely a Fast Fourier Transform (FFT), on a simulated architecture that models the Stanford FLASH multiprocessor and is quite close to the SGI Origin2000 (though much more aggressive in its block transfer capabilities and performance). We can see that the relative benefit of block transfer over ordinary cache-coherent com-
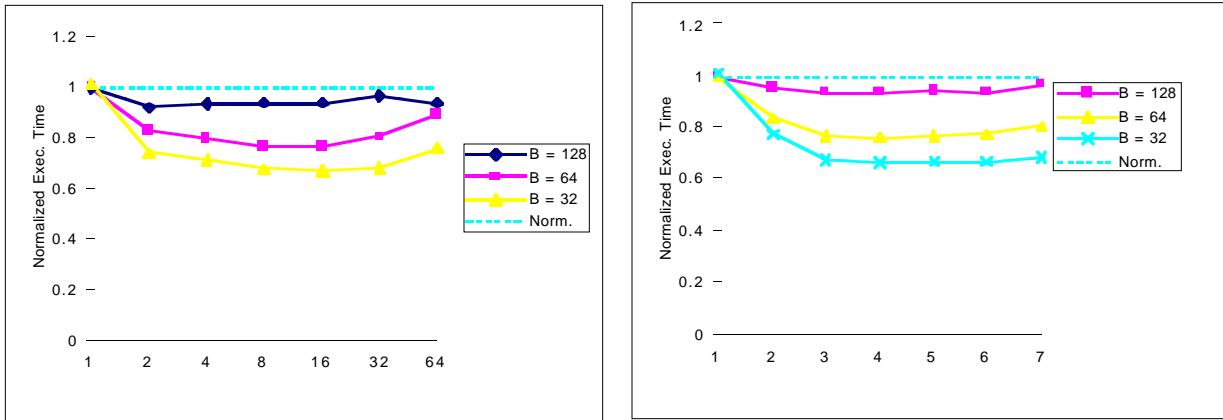
Figure 11-13 The benefits of block data transfer in a Fast Fourier Transform program.

The architectural model and parameters used resemble those of the Stanford Flash multiprocessor [KOH+94], and are similar to those of the SGI Origin2000. Each curve is for a different cache block size. For each curve, the Y axis shows the execution time normalized to that of an execution without block transfer *using the same cache block size*. Thus, the different curves are normalized differently, and different points for the same X-axis value are not normalized to the same number. The closer a point is to the value 1 (the horizontal line), the less the improvement obtained by using block transfer over regular read-write communication for that cache

munication diminishes with increasing cache block size, since the excellent spatial locality in this program causes long cache blocks to themselves behave like little block transfers. For the largest block size of 128 bytes, which is in fact the block size used in the SGI Origin2000, the benefits of using block transfer on such an efficient cache-coherent machine are small, even though the block transfer engine simulated is very optimistic. The sweet-spot effect is clearly visible, and the sweet spot moves with increasing problem size. Results for Ocean, a more complete, regular application whose communication is a more complex variant of the nearest-neighbor communication, are shown are shown in Figure 11-14. Block transfers at row-oriented partition boundaries
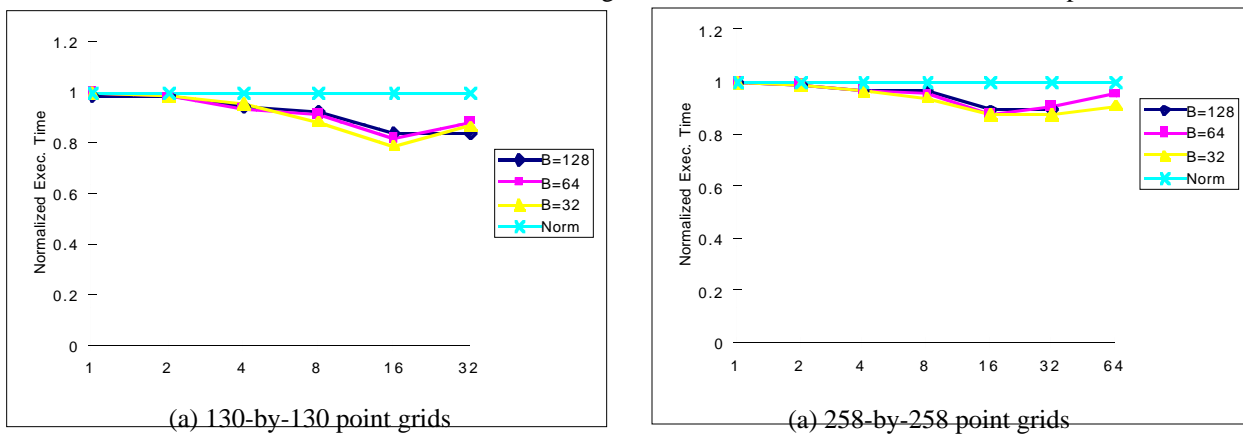
Figure 11-14 The benefits of block transfer in the Ocean application.

The platform and data interpretation are exactly the same as in Figure 11-13.

transfer contiguous data, but this communication also has very good spatial locality; at column-oriented partition boundaries spatial locality in communicated data is poor, but it is also difficult to exploit block transfer well. The relative benefits of block transfer therefore don't depend so

greatly on cache block size. Overall, the communication to computation ratio in Ocean is much smaller than in FFT. While the communication to computation ratio increases at higher levels of the grid hierarchy in the multigrid solver—as processor partitions become smaller—block transfers also become smaller at these levels and are less effective at amortizing overhead. The result is that block transfer is a lot less helpful for this application. Quantitative data for other applications can be found in [WSH94].

As we saw in the Barnes-Hut galaxy simulation in Chapter 3, irregular computations can also be structured to communicate in coarse-grained messages, but this often requires major changes in the algorithm and its orchestration. Often, additional work has to be done to gather the data together for large messages and orient the parallel algorithm that way. Whether this is useful depends on how the extra computation costs compare with the benefits obtained from block transfer. In addition to data needed by the computation directly, block transfer may also be useful for some aspects of parallelism management. For example, in a task stealing scenario if the task descriptors for a stolen task are large they can be sent using a block transfer, as can the associated data, and the necessary synchronization for task queues can be piggybacked on these transfers.

One situation when block transfer will clearly be beneficial is when the overhead to initiate remote read-write accesses is very high, unlike in the architecture simulated above; for example, when the shared address space is implemented in software. To see the effects of other parameters of the communication architecture, such as network transit time and point-to-point bandwidth, let us look at the potential benefits more analytically. In general, the time to communicate a large block of data through remote read misses is (*#of read misses * remote read miss time*), and the time to get the data to the destination processor through block transfer is (*start-up overhead + #of cache blocks transferred * pipeline stage time per line + #of read misses * local read miss time*). The simplest case to analyze is a large enough contiguous chunk of data, so that we can ignore the block transfer start-up cost and also assume perfect spatial locality. In this case, the speedup due to block transfer is limited by the ratio:

$$\frac{remote\ read\ miss\ time}{block\ transfer\ pipeline\ stage\ time + local\ read\ miss\ time} \qquad \textbf{(EQ 11.1)}$$

where the block transfer pipeline stage time is the maximum of the time spent in any of the stage of the pipeline that was shown in Figure 11-10.

A longer network transit time implies that the remote read miss time is increased. If the network bandwidth doesn't decrease with increasing latency, then the other terms are unaffected and longer latencies favor block transfer. Alternatively, network bandwidth may decrease proportionally to the increase in latency e.g. if latency is increased by decreasing the clock speed of the network. In this case, at some point the rate at which data can be pushed into the network becomes smaller than the rate at which the assist can retrieve data from main memory. Up to this point, the memory access time dominates the block transfer pipeline stage time and increases in network latency favor block transfer. After this point the numerator and denominator in the ratio increase at the same rate so the relative advantage of block transfer is unchanged.

Finally, suppose latency and overhead stay fixed but bandwidth changes, (e.g. more wires are added to each network link). We might think that load-based communication is latency bound while block transfer is bandwidth-bound, so increasing bandwidth should favor block transfer. However, past a point increasing bandwidth means that the pipeline stage time becomes domi-

nated by the memory access time, as above, so increasing bandwidth further does not improve the performance of block transfer. Reducing bandwidth increases the block transfer pipeline stage time proportionally, once it starts to dominate the end-point effects, and the extent to which this hurts block transfer depends on how it impacts the end-to-end read miss latency of a cache block. Thus, *if* the other variables are kept constant in each case, block transfer is helped by increased per-message overhead, helped by increased latency, hurt by decreased latency, helped by increased bandwidth up to a point, and hurt by decreased bandwidth.

In summary, then, the performance improvement obtained from block transfer over read-write cache coherent communication depends on the following factors:

- What fraction of the execution time is spent in communication that is amenable to block transfer.
- How much extra work must be done to structure this communication in block transfers.
- The problem size and number of processors, which affect the first point above as well as the sizes of the transfers and thus how well they amortize block transfer overhead.
- The overhead itself, and the different latencies and bandwidths in the system.
- The spatial locality obtained with read-write communication and how it interacts with the granularity of data transfer (cache block size, see Exercise 11.4).

If we can really make all messages large enough, then the latency of communication may not be the problem anymore; bandwidth may become a more significant constraint. But if not, we still need to hide the latency of individual data accesses by overlapping them with computation or with other accesses. The next three sections examine the techniques for doing this in a shared address space. As mentioned earlier, the focus is mostly on the case where communication is performed through reads and writes at the granularity of individual cache blocks or words. However, we shall also discuss how the techniques might be used in conjunction with block transfer. Let us begin with techniques to move past long-latency accesses in the same thread of computation.

11.6 Proceeding Past Long-latency Events in a Shared Address Space

A processor can proceed past a memory operation to other instructions if the memory operation is made *non-blocking*. For writes, this is usually quite easy to implement: The write is put in a write buffer, and the processor goes on while the buffer takes care of issuing the write to the memory system and tracking its completion as necessary. A similar approach can be taken with reads if the processor supports non-blocking reads. However, due to the nature of programs, this requires greater support from software and/or hardware than simply a buffer and the ability to check dependences. The difference is that unlike a write, a read is usually followed very soon by an instruction that needs the value returned by the read (called a dependent instruction). This means that even if the processor proceeds past the read, it will stall very soon since it will reach a dependent instruction. If the read misses in the cache, very little of the miss latency is likely to be hidden in this manner. Hiding read latency effectively, using non-blocking reads, requires that we "look ahead" in the instruction stream past the dependent instruction to find other instructions that are independent of the read. This requires support in the compiler (instruction scheduler) or the hardware or both. Hiding write latency does not usually require instruction lookahead: We can allow the processor to stall when it encounters a dependent instruction, since it is not likely to encounter one soon.

Proceeding past operations before they complete can provide benefits from both buffering and pipelining. Buffering of memory operations allows the processor to proceed with other activity, to delay the propagation and application of requests like invalidations, and to merge operations to the same word or cache block in the buffer before sending them out further into the memory system. Pipelining means that multiple memory operations (cache misses) are issued into the extended memory hierarchy at a time, so their latencies are overlapped and they can complete earlier. These advantages are true for uniprocessors as well as multiprocessors.

Whether or not proceeding past the issuing of memory operations violates sequential consistency in multiprocessors depends on whether the operations are allowed to perform with respect to (become visible to) other processors out of program order. By requiring that memory operations from the same thread should not appear to perform with respect to other processors out of program order, sequential consistency (SC) restricts—but by no means eliminates—the amount of buffering and overlap that can be exploited. More relaxed consistency models allow greater overlap, including allowing some types of memory operations to even complete out of order as discussed in Chapter 9. The extent of overlap possible is thus determined by both the mechanisms provided and the consistency model: Relaxed models may not be useful if the mechanisms needed to support their reorderings (for example, write buffers, compiler support or dynamic scheduling) are not provided, and the success of some types of aggressive implementations for overlap may be restricted by the consistency model.

Since our focus here is latency tolerance, the discussion is organized around increasingly aggressive mechanisms needed for overlapping the latency of read and write operations. Each of these mechanisms is naturally associated with a particular class of consistency models—the class that allows those operations to even complete out of order—but can also be exploited in more limited ways with stricter consistency models, by allowing overlap but not out of order completion. Since hiding read latency requires more elaborate support—albeit provided in many modern microprocessors—to simplify the discussion let us begin by examining mechanisms to hide only write latency. Reads are initially assumed to be blocking, so the processor cannot proceed past them; later, hiding read latency will be examined as well. In each case, we examine the sources of performance gains and the implementation complexity, assuming the most aggressive consistency model needed to fully exploit that overlap, and in some cases also look at the extent to which sequential consistency can exploit the overlap mechanisms. The focus is always on hardware cache-coherent systems, though many of the issues apply quite naturally to systems that don't cache shared data or are software coherent as well.

11.6.1 Proceeding Past Writes

Starting with a simple, statically-scheduled processor with blocking reads, to proceed past write misses the only support we need in the processor is a buffer, for example a write buffer. Writes, such as those that miss in the first-level cache, are simply placed in the buffer and the processor proceeds to other work in the same thread. The processor stalls upon a write only if the write buffer is full.

The write buffer is responsible for controlling the visibility and completion of writes relative to other operations from that processor, so the processor's execution units do not have to worry about this. Similarly, the rest of the extended memory hierarchy can reorder transactions (at least to different locations) as it pleases, since the machinery close to the processor takes care of preserving the necessary orders. Consider overlap with reads. For correct uniprocessor operation, a

read may be allowed to bypass the write buffer and be issued to the memory system, as long as there is not a write to the same location pending in the write buffer. If there is, the value from the write buffer may be forwarded to the read even before the write completes. The latter mechanism will certainly allow reads to complete out of order with respect to earlier writes in program order, thus violating SC, unless the write buffer is flushed upon a match and the read reissued to the memory system thereafter. Reads bypassing the write buffer will also violate SC, unless the read is not allowed to complete (bind its value) before those writes. Thus, SC can take advantage of write buffers, but the benefits are limited. We know from Chapter 9 that allowing reads to complete before previous writes leads to models like processor consistency (PC) and total store ordering (TSO).

Now consider the overlap among writes themselves. Multiple writes may be placed in the write buffer, and the order in which they are made visible or they complete is also determined by the buffer. If writes are allowed to complete out of order, a great deal of flexibility and overlap among writes can be exploited by the buffer. First, a write to a location or cache block that is currently anywhere in the buffer can be merged (coalesced) into that cache block and only a single ownership request sent out for it by the buffer, thus reducing traffic as well. Especially if the merging is not into the last entry of the buffer, this clearly leads to writes becoming visible out of program order. Second, buffered writes can be retired from the buffer as soon as possible, before they complete, making it possible for other writes behind them to get through. This allows the ownership requests of the writes to be pipelined through the extended memory hierarchy.

Allowing writes to complete out of order requires partial store order (PSO) or a more relaxed consistency model. Stricter models like SC, TSO and PC essentially require that merging into the write buffer not be allowed except into the last block in the buffer, and even then in restricted circumstances since other processors must not be able to see the writes to different words in the block in a different order. Retiring writes early to let others pass through is possible under the stricter models as long as the order of visibility and completion among these writes is preserved in the rest of the system. This is relatively easy in a bus-based machine, but very difficult in a distributed-memory machine with different home memories and independent network paths. On the latter systems, guaranteeing program order among writes, as needed for SC, TSO and PC, essentially requires that a write not be retired from the head of the (FIFO) buffer until it has committed with respect to all processors.

Overall, a strict model like SC can utilize write buffers to overlap write latency with reads and other writes, but only to a limited extent. Under relaxed models, when exactly write operations are sent out to the extended memory hierarchy and made visible to other processors depends on implementation and performance considerations as well. For example, under a relaxed model invalidations may be sent out as soon as the writes are able to get through the write buffer, or the writes may be delayed in the buffer until the next synchronization point. The latter may allow substantial merging of writes, as well as reduction of invalidations and misses due to false sharing. However, it implies more traffic in bursts at synchronization points, rather than pipelined traffic throughout the computation.

Performance Impact

Simulation results are available in the literature [GGH91a,Gha95] that measure the benefits of allowing the processor to proceed past writes on the parallel applications in the original SPLASH application suite [SWG92]. They do not try to maintain SC, but exploit the maximum overlap

that is possible without too much extra hardware. They are separated into techniques that allow the program order between a write and a following read (write->read order) to be violated, satisfying TSO or PC, and those that also allow the order between two writes (write->write order) to be violated, satisfying PSO. The latter is also essentially the best that more relaxed models like RMO, WO or RC can accomplish given that reads are blocking. Let us look at these results in the rest of this subsection.

Figure 11-15 shows the reduction in execution time, divided into different components, for two representative applications when the write->read and write->write orders order is allowed to be violated. The applications are older versions of Ocean and Barnes-Hut applications, running with small problem sizes on 16-processor, simulated, directory-based cache-coherent systems. The baseline for comparison is the straightforward implementation of sequential consistency, in which the processor issuing a read or write reference stalls until that reference completes (measurements for using the write buffer but preserving SC—by stalling a read until the write buffer is empty—showed only a very small performance improvement over stalling the processor on all writes themselves). Details of the simulated system, which assumes considerably less aggressive processors than modern systems, and an analysis of the results can be found in [GGH91a,Gha95]; here, we simply present a few of the results to illustrate the effects.
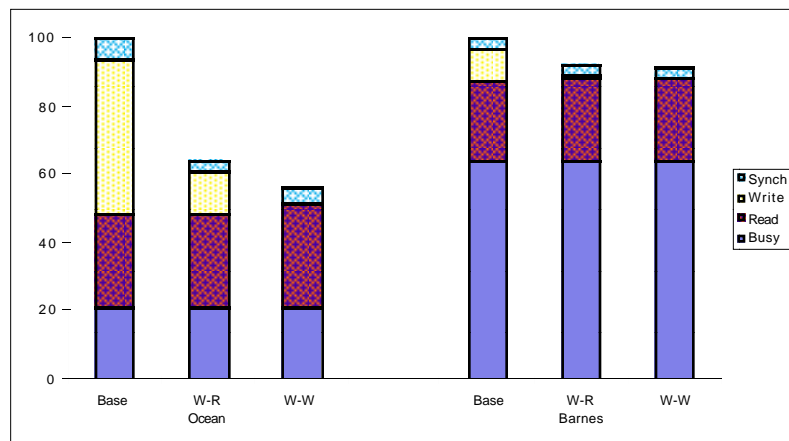
[Artist: please change "Barnes" to "Barnes-Hut" in this and other figures.]

Figure 11-15 Performance benefits from proceeding past writes by taking advantage of relaxed consistency models.

The results assume a statically scheduled processor model. For each application, Base is the case with no latency hiding, W-R indicates that reads can bypass writes (i.e. the write->read order can be violated, and W-W indicates that both writes and reads can bypass writes. The execution time for the Base case is normalized to 100 units. The system simulated is a cache-coherent multiprocessor using a flat, memory-based directory protocol much like that of the Stanford DASH multiprocessor [LLJ+92]. The simulator assumes a write-through first-level and write-back second-level caches, with a deep, 16-entry write buffer between the two. The processor is single-issue and statically scheduled, with no special scheduling in the compiler targeted toward latency tolerance. The write buffer is aggressive, with read bypassing of it and read forwarding from it enabled. To preserve the write-write order (initially), writes are not merged in the write buffer, and the write at the head of the FIFO write buffer is not retired until it has been completed. The data access parameters assumed for a read miss are one cycle for an L1 hit, 15 cycles for an L2 hit, 29 cycles if satisfied in local memory, 101 cycles for a two-message remote miss, and 132 cycles for a three-message remote miss. Write latencies are somewhat smaller in each case. The system therefore assumes a much slower processor relative to a memory system than modern systems.

The figure shows that simply allowing write->read reordering is usually enough to hide most of the write latency from the processor (see the first two bars for each application). All the applications in the study had their write latency almost entirely hidden by this deep write buffer, except Ocean which has the highest frequency of write misses to begin with. There are also some interesting secondary effects. One of these is that while we might not expect the read stall time to change much from the base SC case, in some cases it increases slightly. This is because the addi-

tional bandwidth demands of hiding write latency contend with read misses, making them cost more. This is a relatively small effect with these simple processors, though it may be more significant with modern, dynamically scheduled processors that also hide read latency. The other, beneficial effect is that synchronization wait time is sometimes also reduced. This happens for two reasons. First some of the original load imbalance is caused by imbalances in memory stall time; as stall time is reduced, imbalances are reduced as well. Second, if a write is performed inside a critical section and its latency is hidden, then the critical section completes more quickly and the lock protecting it can be passed more quickly to another processor, which therefore incurs less wait time for that lock.

The third bar for each application in the figure shows the results for the case when the write-write order is allowed to be violated as well. Now write merging is enabled to any block in the same 16-entry write buffer, and a write is allowed to retire from the write buffer as soon as it reaches the head, even before it is completed. Thus, pipelining of writes through the memory and interconnect is given priority over buffering them for more time, which might have enabled greater merging in the write buffer and reduction of false sharing misses by delaying invalidations. Of course, having multiple writes outstanding in the memory system requires that the caches allow multiple outstanding misses, i.e. that they are lockup-free. The implementation of lockup-free caches will be discussed in Section 11.9.

The figure shows that the write-write overlap hides whatever write latency remained with only write-read overlap, even in Ocean. Since writes retire from the write buffer at a faster rate, the write buffer does not fill up and stall the processor as easily. Synchronization wait time is reduced further in some cases, since a sequence of writes before a release completes more quickly causing faster transfer of synchronization between processors. In most applications, however, write-write overlap doesn't help too much since almost all the write latency was already hidden by allowing reads to bypass previous writes. This is especially true because the processor model assumes that reads are blocking, so write-write overlap cannot be exploited past read operations. For reordering by hardware on cache-coherent systems, differences between models like weak ordering and release consistency—as well as subtle differences among models that allow the same reorderings, such as TSO which preserves write atomicity and processor consistency which does not—do not seem to affect performance substantially.

Since performance is highly dependent on implementation and on problem and cache sizes, the study examined the impact of using less aggressive write buffers and second-level cache architectures, as well as of scaling down the cache size to capture the case where an important working set does not fit in the cache (see Chapter 4). For the first part, four different combinations were examined: an L2 cache that blocks on a miss versus a lockup-free L2 cache, and a write buffer that allows reads to bypass it if they are not to an address in the buffer and one that does not. The results indicate that having a lockup-free cache is very important to obtaining good performance improvements, as we might expect. As for bypassable write-buffers, they are very important for the system that allows write->read reordering, particularly when the L2 cache is lockup-free, but less important for the system that allows write->write reordering as well. The reason is that in the former case writes retire from the buffer only after completion, so it is quite likely that when a read misses in the first-level cache it will find a write still in the write buffer, and stalling the read till the write buffer empties will hurt performance; in the latter case, writes retire much faster so the likelihood of finding a write in the buffer to bypass is smaller. For the same reason, it is easier to use smaller write buffers effectively when write->write reordering is allowed so writes are allowed to retire early. Without lockup-free caches, the ability for reads to bypass writes in the buffer is much less advantageous.

The results with smaller L1/L2 caches are shown in Figure 11-16, as are results for varying the

(a) Effects of Cache Size

(b) Effects of Cache Block Size for Ocean
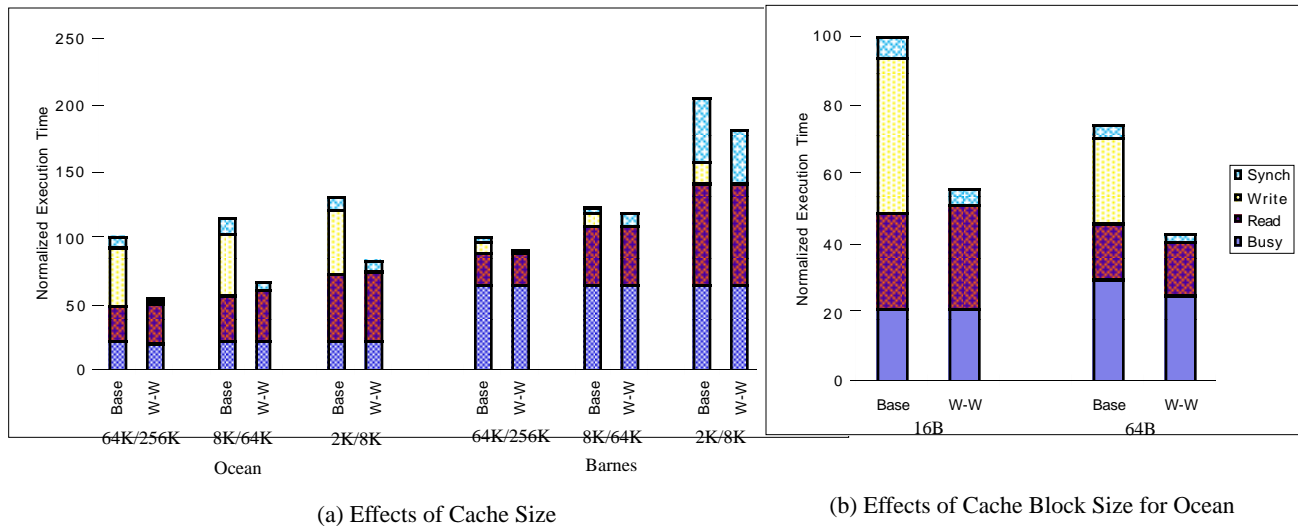
Figure 11-16 Effects of cache size and cache block size on the benefits of proceeding past writes.

The block size assumed in the graph for varying cache size is the default 16B used in the study. The cache sizes assumed in the graph for varying block size are the default 64KB first level and 256KB second level caches.

cache block size. Consider cache size first. With smaller caches, write latency is still hidden effectively by allowing write->read and write->write reordering. (As discussed in Chapter 4, for Barnes-Hut the smallest caches do not represent a realistic scenario.) Somewhat unexpectedly, the impact of hiding write latency is often larger with larger caches, even though the write latency to be hidden is smaller. This is because larger caches are more effective in reducing read latency than write latency, so the latter is relatively more important. All the above results assumed a cache block size of only 16 bytes, which is quite unrealistic today. Larger cache blocks tend to reduce the miss rates for these applications and hence reduce the impact of these reorderings on execution time a little, as seen for Ocean in Figure 11-16(b).

11.6.2 Proceeding Past Reads

As discussed earlier, to hide read latency effectively we need not only non-blocking reads but also a mechanism to look ahead beyond dependent instructions. The later independent instructions may be found by the compiler or by hardware. In either case, implementation is complicated by the fact that the dependent instructions may be followed soon by branches. Predicting future paths through the code to find useful instructions to overlap requires effective branch prediction, as well as speculative execution of instructions past predicted branches. And given speculative execution, we need hardware support to cancel the effects of speculatively executed instructions upon detecting misprediction or poor speculation.

The trend in the microprocessor industry today is toward increasingly sophisticated processors that provide all these features in hardware. For example, they are included by processor families such as the Intel Pentium Pro [Int96], the Silicon Graphics R10000 [MIP96], the SUN Ultrasparc [LKB91], and the HP-PA8000 [Hun96]. The reason is that latency hiding and overlapped opera-

tion of the memory system and functional units are increasingly important even in uniprocessors. These mechanisms have a high design and hardware cost, but since they are already present in the microprocessors they can be used to hide latency in multiprocessors as well. And once they are present, they can be used to hide both write latency as well as read latency.

Dynamic Scheduling and Speculative Execution

To understand how memory latency can be hidden by using dynamic scheduling and speculative execution, and especially how the techniques interact with memory consistency models, let us briefly recall from uniprocessor architecture how the methods work. More details can be found in the literature, including [HeP95]. Dynamic scheduling means that instructions are decoded in the order presented by the compiler, but they are executed by their functional units in the order in which their operands become available at runtime (thus potentially in a different order than program order). One way to orchestrate this is through reservation stations and Tomasulo's algorithm [HeP95]. This does not necessarily mean that they will become visible or complete out of program order, as we will see. Speculative execution means allowing the processor to look at and schedule for execution instructions that are not necessarily going to be useful to the program's execution, for example instructions past a future uncertain branch instruction. The functional units can execute these instructions assuming that they will indeed be useful, but the results are not committed to registers or made visible to the rest of the system until the validity of the speculation has been determined (e.g. the branch has been resolved).

The key mechanism used for speculative execution is an instruction *lookahead* or *reorder* buffer. As instructions are decoded by the decode unit, whether in-line or speculatively past predicted branches, they are placed in the reorder buffer, which therefore holds instructions in program order. Among these, any instructions that are not dependent on a yet incomplete instruction can be chosen from the reorder buffer for execution. It is quite possible that independent long latency instructions (e.g. reads) that are before them in the buffer have not yet been executed or completed, so in this way the processor thus proceeds past incomplete memory operations and other instructions. The reorder buffer implements register renaming, so instructions in the buffer become ready for execution as soon as their operands are produced by other instructions, not necessarily waiting till the operands are available in the register file. The results of an instruction are maintained with it in the reorder buffer, and not yet committed to the register file. An instruction is retired from the reorder buffer only when it reaches the head of the reorder buffer, i.e. in program order. It is at this point that the result of a read may be put in the register and the value produced by a write is free to be made visible to the memory system.

Retiring in program (decode) order makes it easy to perform speculation: if a branch is found to be mispredicted, then no instruction that was after it in the buffer (hence in program order) can have left the reorder buffer and committed its effects: No read has updated a register, and no write has become visible to the memory system. All instructions after the branch are then invalidated from the reorder buffer and the reservations stations, and decoding is resumed from the correct (resolved) branch target. In-order retirement also makes it easy to implement precise exceptions, so the processor can restart quickly after an exception or interrupt. However, in-order retirement does mean that if a read miss reaches the head of the buffer before its data value has come back, the reorder buffer (not the processor) stalls since later instructions cannot be retired. This FIFO nature and stalling implies that the extent of overlap and latency tolerance may be limited by the size of the reorder buffer.

Hiding Latency under Sequential and Release Consistency

The difference between retiring and completion is important. A read completes when its value is bound, which may be before it reaches the head of the reorder buffer and can retire (modifying the processor register). On the other hand, a write is not complete when it reaches the head of the buffer and retires to the memory system; it completes only when it has actually become visible to all processors. Understand this difference helps us understand how out-of-order, speculative processors can hide latency under both SC and more relaxed models like release consistency (RC).

Under RC, a write may indeed be retired from the buffer before it completes, allowing operations behind it to retire more quickly. A read may be issued and complete anytime once it is in the reorder buffer, unless there is an acquire operation before it that has not completed. Under SC, a write is retired from the buffer only when it has completed (or at least committed with respect to all processors, see Chapter 6), so it may hold up the reorder buffer for longer once it reaches the head and is sent out to the memory system. And while a read may still be issued to the memory system and complete before it reaches the head of the buffer, it is not issued before all previous memory operations in the reorder buffer have completed. Thus, SC again exploits less overlap than RC.

Additional Techniques to Enhance Latency Tolerance

Additional techniques—like a form of hardware prefetching, speculative reads, and write buffers— can be used with these dynamically scheduled processors to hide latency further under SC as well as RC [KGH91b], as mentioned in Chapter 9 (Section 9.2). In the hardware prefetching technique, hardware issues prefetches for any memory operations that are in the reorder buffer but are not yet permitted by the consistency model to actually issue to the memory system. For example, the processor might prefetch a read that is preceded by another incomplete memory operation in SC, or by an incomplete acquire operation in RC, thus overlapping them. For writes, prefetching allows the data and ownership to be brought to the cache before the write actually gets to the head of the buffer and can be issued to the memory system. These prefetches are *non-binding*, which means that the data are brought into the cache, not into a register or the reorder buffer, and are still visible to the coherence protocol, so they do not violate the consistency model. (Prefetching will be discussed in more detail in Section 11.7.)

Hardware prefetching is not sufficient when the address to be prefetched is not known, and determining the address requires the completion of a long-latency operation. For example, if a read miss to an array entry A[I] is preceded by a read miss to I, then the two cannot be overlapped because the processor does not know the address of A[I] until the read of I completes. The best we could do is prefetch I, and then issue the read of A[I] before the actual read of I completes. However, suppose now there is a cache miss to a different location X between the reads of I and A[I]. Even if we prefetched I, under sequential consistency we cannot execute and complete the read of A[I] before the read of X has completed, so we have to wait for the long-latency miss to X. To increase the overlap, the second technique that a processor can use is *speculative read* operations. These read operations use the results of previous operations (often prefetches) that have not yet completed as the addresses to read from, speculating that the value used as the address will not change between now and the time when all previous operations have completed. In the above example, we can use the value prefetched or read for I and issue, execute and complete the read of A[I] before the read of X completes.

What we need to be sure of is that we used the correct value for I and hence read the correct value for A[I], i.e. that they are not modified between the time of the speculative read and the time when it is possible to perform the actual read according to the consistency model. For this reason, speculative reads are loaded into not the registers themselves but a buffer called the speculative read buffer. This buffer "watches" the cache and hence is apprised of cache actions to those blocks. If an invalidation, update or even cache replacement occurs on a block whose address is in the speculative read buffer before the speculatively executed read has reached the front of the reorder buffer and retired, then the speculative read and all instructions following it in the reorder buffer must be cancelled and the execution rolled back just as on a mispredicted branch. Such hardware prefetching and speculative read support are present in processors such as the Pentium Pro [Int96], the MIPS R10000 [MIP96] and the HP-PA8000 [Hun96]. Note that under SC, every read issued before previous memory operations have completed is a speculative read and goes into the speculative read buffer, whereas under RC reads are speculative (and have to watch the cache) only if they are issued before a previous acquire has completed.

A final optimization that is used to increase overlap is the use of a write buffer even in a dynamically scheduled processor. Instead of writes having to wait at the head of the reorder buffer until they complete, holding up the reorder buffer, they are removed from the reorder buffer and placed in the write buffer. The write buffer allows them to become visible to the extended memory hierarchy and keeps track of their completion as required by the consistency model. Write buffers are clearly useful with relaxed models like RC and PC, in which reads are allowed to bypass writes: The reads will now reach the head of the write buffer and retire more quickly. Under SC, we can put writes in the write buffer, but a read stalls when it reaches the head of the reorder buffer anyway until the write completes. Thus, it may be that much of the latency that the store would have seen is not hidden but is instead seen by the next read to reach the head of the reorder buffer.

Performance Impact

Simulation studies in the literature have examined the extent to which these techniques can hide latency under different consistency models and different combinations of techniques. The studies do not use sophisticated compiler technology to schedule instructions in a way that can obtain increased benefits from dynamic scheduling and speculative execution, for example by placing more independent instructions close after a memory operation or miss so smaller reorder buffers will suffice. A study assuming an RC model finds that a substantial portion of read latency can indeed be hidden using a dynamically scheduled processor with speculative execution (let's call this a DS processor), and that the amount of read latency that can be hidden increases with the size of the reorder buffer even up to buffers as large as 64 to 128 entries (the longer the latency to be hidden, the greater the lookahead needed and hence the larger the reorder buffer) [KGH92]. A more detailed study has compared the performance of the SC and RC consistency models with aggressive, four-issue DS processors [PRA+96]. It has also examined the benefits obtained individually from hardware prefetching and speculative loads in each case. Because a write takes longer to complete even on an L1 hit when the L1 cache is write-through rather than write-back, the study examined both types of L1 caches, always assuming a write-back L2 cache. While the study uses very small problem sizes and scaled down caches, the results shed light on the interactions between mechanisms and models.

The most interesting question is whether with all these optimizations in DS processors, RC still buys substantial performance gains over SC. If not, the programming burden of relaxed consistency models may not be justified with modern DS processors (relaxed models are still important
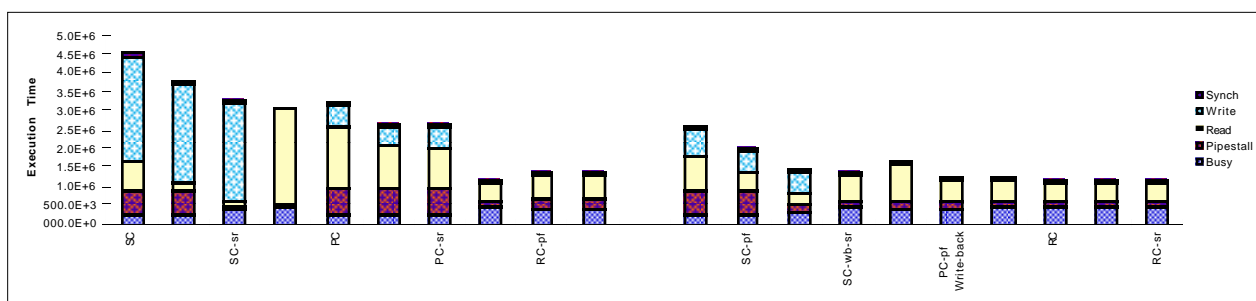
Figure 11-17 Performance of an FFT kernel with different consistency models, assuming a dynamically scheduled processor with speculative execution.

The bars on the left assume write-through first-level caches, and the bars on the right assume write-back first-level caches. Second-level caches are always write-back. SC, PC and RC are sequential, processor, and release consistency, respectively. For SC, there are two sets of bars: the first assuming no write buffer and the second including a write buffer. For each set of three bars, the first bar excludes hardware prefetching and speculative loads, the second bar (PF) includes the use of hardware prefetching, and the third bar (SR) includes the use of speculative read operations as well. The processor model assumed resembles the MIPS R10000 [MIP96]. The processor is clocked at 300MHz and capable of issuing 4 instructions per cycle. It used a reorder buffer of size 64 entries, a merging write buffer with 8 entries, a 4KB direct-mapped first-level cache and a 64 KB, 4-way set associative second level cache. Small caches are chosen since the data sets are small, and they may exaggerate the effect of latency hiding. More detailed parameters can be found in [PRA+96].

at the programmer's interface to allow compiler optimizations, but the contract and the programming burden may be lighter if it is only the compiler that may reorder operations). The results of the study, shown for two programs in Figures 11-17 and 11-18, indicate that RC is still beneficial, even though the gap is closed substantially. The figures show the results for SC without any of the more sophisticated optimizations (hardware prefetching, speculative reads, and write buffering), and then with those applied cumulatively one after the other. Results are also shown for processor consistency (PC) and for RC. These cases always assume write buffering, and are shown both without the other two optimizations and then with those applied cumulatively.
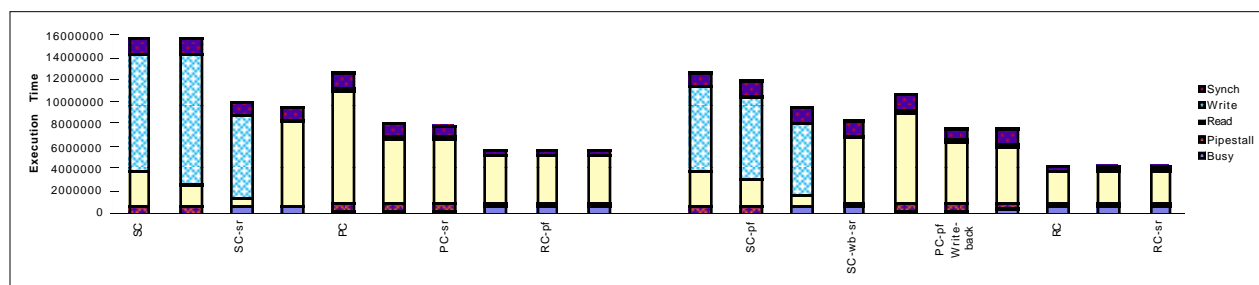
Figure 11-18 Performance of Radix with different consistency models, assuming a dynamically scheduled processor with speculative execution.

The system assumptions and organization of the figure are the same as in Figure 11-17.

Clearly, the results show that RC has substantial advantages over SC even with a DS processor when hardware prefetching and speculative reads are not used. The main reason is that RC is able to hide write latency much more successfully than SC, as with simple non-DS processors, since it allows writes to be retired faster and allows later accesses like reads to be issued and complete

before previous writes. The improvement due to RC is particularly true with write-through caches, since under SC a write that reaches the head of the buffer issues to the memory system but has to wait until the write performs in the second-level cache, holding up later operations, while under RC it can retire when it issues and before it completes. While read latency is hidden with some success even under SC, RC allows for much earlier issue, completion and hence retirement of reads. The effects of hardware prefetching and speculative reads are much greater for SC than for RC, as expected from the discussion above (since in RC operations are already allowed to issue and complete out of order, so the techniques only help those operations that follow closely after acquires). However, a significant gap still remains compared to RC, especially in write latency. Write buffering is not very useful under SC for the reason discussed above.

The figure also confirms that write latency is a much more significant problem with write-through first-level caches under SC than with write-back caches, but is hidden equally well under RC with both types of caches. The difference in write latency between write-through and write-back caches under SC is much larger for FFT than for Radix. This is because while in FFT the writes are to locally allocated data and hit in the write-back cache for this problem size, in Radix they are to nonlocal data and miss in both cases. Overall, PC is in between SC and RC, with hardware prefetching helping but speculative reads not helping so much as in SC.

The graphs also appear to reveal an anomaly: The processor busy time is different in different scheme, even for the same application, problem size and memory system configuration. The reason is an interesting methodological point for superscalar processors: since several (here four) instructions can be issued every cycle, how do we decide whether to attribute a cycle to busy time or a particular type of stall time. The decision made in most studies in the literature, and in these results, is to attribute a cycle to busy time only if all instruction slots issue in that cycle. If not, then the cycle is attributed to whatever form of stall is seen by the first instruction (starting from the head of the reorder buffer) that should have issued in that cycle but didn't. Since this pattern changes with consistency models and implementations, the busy time is not the same across schemes.

While the main results have already been discussed, it is interesting to examine the interactions with hardware prefetching and speculative reads in a little more detail, particularly in how they interact with application characteristics. The benefits from this form of hardware prefetching are limited by several effects. First, prefetches are issued only for operations that are in the reorder buffer, so they cannot be issued very far in advance. Prefetching works much more successfully when a number of operations that will otherwise miss are bunched up together in the code, so they can all be issued together from the reorder buffer. This happens naturally in the matrix transposition phase of an FFT, so the gains from prefetching are substantial, but it can be aided in other programs by appropriate scheduling of operations by the compiler. Another limitation is the one that motivated speculative loads in the first place: The address to be prefetched may not be known, and the read that returns it may be blocked by other operations. This effect is encountered in the Radix sorting application, shown in Figure 11-18. Radix also encounters misses close to each other in time, but the misses (both write misses to the output array in the permutation phase and read misses in building the global histograms) are to array entries indexed by histogram values that have to be read as well.

Speculative loads help the performance of Radix tremendously, as expected, improving both read and write miss stall times in this case. An interesting effect in both programs is that the processor busy time is reduced as well. This is because the ability to consume the values of reads speculatively makes a lot more otherwise dependent instructions available to execute, greatly increasing

the utilization of the 4-way superscalar processor. We see that speculative loads help reduce read latency in FFT as well (albeit less), even though it does not have indirect array accesses. The reason is that conflict misses in the cache are reduced due to greater combining of accesses to an outstanding cache block (accesses to blocks that conflict in the cache and would otherwise have caused conflict misses are overlapped and hence combined in the mechanism used to keep track of outstanding misses). This illustrates that sometimes important observed effects are not directly due to the feature being studied. While speculative reads, and speculative execution in general, can be hurt by mis-speculation and consequent rollbacks, this hardly occurs at all in these programs, which take quite predictable and straightforward paths through the code. The results may be different for programs with highly unpredictable control flow and access patterns. (Rollbacks are even fewer under RC, where there is less need for speculative loads.)

11.6.3 Implementation Issues

To proceed past writes, the only support we need in the processor is a write buffer and nonblocking writes. Increasing sophistication in the write buffer (merging to arbitrary blocks, early retirement before obtaining ownership) allows both reads and writes to effectively bypass previous writes. To proceed effectively past reads, we need not just nonblocking reads but also support for instruction lookahead and speculative execution. At the memory system level, supporting multiple outstanding references (particularly cache misses) requires that caches be lockup-free. Lockup-free caches also find it easier to support multiple outstanding write misses (they are simply buffered in a write buffer) than to support multiple outstanding read misses: For reads they also need to keep track of where the returned data is to be placed back in the processor environment (see the discussion of lockup-free cache design in Section 11.9). In general, once this machinery close to the processor takes care of tracking memory operations and preserving the necessary orders of the consistency model, the rest of the memory and interconnection system can reorder transactions related to different locations as it pleases.

In previous chapters we have discussed implementation issues for sequential consistency on both bus-based and scalable multiprocessors. Let us briefly look here at what other mechanisms we need to support relaxed consistency models; in particular, what additional support is needed close to the processor to constrain reorderings and satisfy the consistency model, beyond the uniprocessor mechanisms and lockup-free caches. Recall from Chapter 9 that the types of reorderings allowed by the processor have affected the development of consistency models; for example, models like processor consistency, TSO and PSO were attractive when processors did not support non-blocking reads. However, compiler optimizations call for more relaxed models at the programmer's interface regardless of what the processor supports, and particularly with more sophisticated processors today the major choice seems to be between SC and models that allow all forms of reordering between reads and writes between synchronizations (weak ordering, release consistency, SUN RMO, DEC Alpha, IBM PowerPC).

The basic mechanism we need is to keep track of the completion of memory operations. But this was needed in SC as well, even for the case of a single outstanding memory operation. Thus, we need explicit acknowledgment transactions for invalidations/updates that are sent out, and a mechanism to determine when all necessary acknowledgments have been received. For SC, we need to determine when the acknowledgments for each individual write have arrived; for RC, we need only determine at a release point that all acknowledgments for previous writes have arrived. For different types of memory barriers or fences, we need to ensure that all relevant operations before the barrier or fence have completed. As was discussed for SC in the context of the SGI

Origin2000 case study in Chapter 8, a processor usually expects only a single response to its write requests, so coordinating the receipt of data replies and acknowledgments is usually left to an external interface.

For example, to implement a memory barrier or a release, we need a mechanism to determine when all previously issued writes and/or reads have completed. Conceptually, each processor (interface) needs a counter that keeps track of the number of uncompleted reads and writes issued by that processor. The memory barrier or release operation can then check whether or not the appropriate counters are zero before allowing further memory references to be issued by that processor. For some models like processor consistency, in which writes are non-atomic, implementing memory barrier or fence instructions can become quite complicated (see [GLL+90, Gha95] for details).

Knowledge of FIFO orderings in the system buffers can allow us to optimize the implementation of some types of fences or barriers. An example is the write-write memory barriers (also called store barriers) used to enforce write-to-write order in the PSO model when using a write buffer that otherwise allows writes to be reordered. On encountering such a MEMBAR, it is not necessary to stall the processor immediately to wait for previous writes to complete. The MEMBAR instruction can simply be inserted into the write-buffer (like any other write), and the processor can continue on and even execute subsequent write operations. These writes are inserted behind the MEMBAR in the write buffer. When the MEMBAR reaches the head of the write buffer, it is at this time that the write buffer stalls (not necessarily the processor itself) to wait for all previous stores to complete before allowing any other stores to become visible to the external memory system. A similar technique can be used with reorder buffers in speculative processors.

Of course, one mechanism that is needed in the relaxed models is support for the order-preserving instructions like memory barriers, fences, and instructions labeled as synchronization, acquire or release. Finally, for most of the relaxed models we need mechanisms to ensure write atomicity. This, too was needed for SC, and we simply need to ensure that the contents of a memory block for which invalidation acknowledgments are pending are not given to any new processor requesting them until the acknowledgments are received. Thus, given the support to proceed past writes and/or reads provided by the processor and caches, most of the mechanisms needed for relaxed models are already needed to satisfy the sufficient conditions for SC itself.

While the implementation requirements are quite simple, they inevitably lead to tradeoffs. For example, for counting acknowledgments the location of the counters can be quite critical to performance. Two reasonable options are: (i) close to the memory controller, and (ii) close to the first-level cache controller. In the former case, updates to the counters upon returning acknowledgments do not need to propagate up the cache hierarchy to the processor chip; however, the time to execute a memory barrier itself becomes quite large because we have to traverse the hierarchy cross the memory bus to query the counters. The latter option has the reverse effect, with greater overhead for updating counter values but lower overhead for fence instructions. The correct decision can be made on the basis of the frequency with which fences will be used relative to the occurrence of cache misses. For example, if we use fences to implement a strict model like SC on a machine which does not preserve orders (like the DEC Alpha), we expect fences to be very frequent and the option of placing the counters near the memory controller to perform poorly.[1]

11.6.4 Summary

The extent to which latency can be tolerated by proceeding past reads and writes in a multiprocessor depends on both the aggressiveness of the implementation and the memory consistency model. Tolerating write latency is relatively easy in cache-coherent multiprocessors, as long as the memory consistency model is relaxed. Both read and write latency can be hidden in sophisticated modern processors, but still not entirely. The instruction lookahead window (reorder buffer) sizes needed to hide read latency can be substantial, and grow with the latency to be hidden. Fortunately, latency is increasingly hidden as window size increases, rather than needing a very large threshold size before any significant latency hiding takes place. In fact, with these sophisticated mechanisms that are widely available in modern multiprocessors, read latency can be hidden quite well even under sequential consistency. In general, conservative design choices—such as blocking reads or blocking caches—make preserving orders easier, but at the cost of performance. For example, delaying writes until all previous instructions are complete in dynamically scheduled processors to avoid rolling back on writes (e.g. for precise exceptions) makes it easier to preserve SC, but it makes write latency more difficult to hide under SC.

Compiler scheduling of instructions can help dynamically scheduled processors hide read latency even better. Sophisticated compiler scheduling can also help statically scheduled processors (without hardware lookahead) to hide read latency by increasing the distance between a read and its dependent instructions. The implementation requirements for relaxed consistency models are not large, beyond the requirements already needed (and provided) to hide write and read latency in uniprocessors.

The approach of hiding latency by proceeding past operations has two significant drawbacks. The first is that it may very well require relaxing the consistency model to be very effective, especially with simple statically scheduled processors but also with dynamically scheduled ones. This places a greater burden on the programmer to label synchronization (competing) operations or insert memory barriers, although relaxing the consistency model is to a large extent needed to allow compilers to perform many of their optimizations anyway. The second drawback is the difficulty of hiding read latency effectively with processors that are not dynamically scheduled, and the resource requirements of hiding multiprocessor read latencies with dynamically scheduled processors. In these situations, other methods like precommunication and multithreading might be more successful at hiding read latency.

11.7 Precommunication in a Shared Address Space

Precommunication, especially prefetching, is a technique that has already been widely adopted in commercial microprocessors, and its importance is likely to increase in the future. To understand

---

1. There are many other options too. For example, we might have a special signal pin on the processor chip that is connected to the external counter and indicates whether or not the counter is zero (to implement the memory barrier we only need to know if the counter is zero; not its actual value.) However, this can get quite tricky. For example, how does one ensure that there are no memory accesses issued before the memory barrier that are still propagating from the processor to the memory and that the counter does not yet even know about. When querying the external counter directly this can be taken care of by FIFO ordering of transactions to and from the memory; with a shortcut pin, it becomes difficult.

the techniques for pre-communication, let us first consider a shared address space with no caching of shared data, so all data accesses go to the relevant main memory. This will introduce some basic concepts, after which we shall examine prefetching in a cache-coherent shared address space, including performance benefits and implementation issues.

11.7.1 Shared Address Space With No Caching of Shared Data

In this case, receiver-initiated communication is triggered either by reads of nonlocally allocated data, and sender-initiated by writes to nonlocally allocated data. In our example code in Figure 11-1 on page 755, the communication is sender initiated, if we assume that the array A is allocated in the local memory of process P_B, not P_A. As with the sends in the message-passing case, the most precommunication we can do is perform all the writes to A before we compute any of the $f(B[])$, by splitting into two loops (see the left hand side of Figure 11-19). By making the writes non blocking, some of the write latency may be overlapped with the $(B[])$ computations; however, much of the write latency would probably have been hidden anyway if writes were made non blocking and left where they were. The more important effect is to have the writes and the overall computation on P_A complete earlier, hence allowing the reader P_B, to emerge from its while loop more quickly.

If the array A is allocated in the local memory of P_A, the communication is receiver-initiated. The writes by the producer P_A are now local, and the reads by the consumer P_B are remote. Precommunication in this case means *prefetching* the elements of A before they are actually needed, just as we issued a_receive's before they were needed in the message-passing case. The difference is that the prefetch is not just posted locally, like the receive, but rather causes a prefetch request to be sent across the network to the remote node (P_A) where the controller responds to the request by actually transferring the data back. There are many ways to implement prefetching, as we shall see. One is to issue a special *prefetch* instruction, and build a software pipeline as in the message passing case. The shared address space code with prefetching is shown in Figure 11-19. The software pipeline has a prologue that issues a prefetch for the first iteration, a steady-state

```
        Process P_A                          Process P_B

                                     while flag = 0 {};
for i←0to n-1 do                     prefetch(A[0]);
   compute A[i];                     for i←0 to n-2 do
   write A[i];                          prefetch (A[i+1]);
end for                                 read A[i] from prefetch_buffer;
flag ← 1;                               use A[i];
for i←0to n-1 do                        compute g(C[i]);
   compute f(B[i]);                  end for
                                     read A[n-1] from prefetch_buffer;
                                     use A[n-1];
                                     compute g(C[n-1])
```

Figure 11-19 Prefetching in the shared address space example.

period of *n-1* iterations in which a prefetch is issued for the next iteration and the current iteration is executed, and an epilogue consisting of the work for the last iteration. Note that a prefetch instruction does not replace the actual read of the data item, and that the prefetch instruction itself must be nonblocking (must not stall the processor) if it is to achieve its goal of hiding latency through overlap.

Since shared data are not cached in this case, the prefetched data are brought into a special hardware structure called a prefetch buffer. When the word is actually read into a register in the next iteration, it is read from the head of the prefetch buffer rather than from memory. If the latency to hide were much larger than the time to compute a single loop iteration, we would prefetch several iterations ahead and there would potentially be several words in the prefetch buffer at a time. However, if we can assume that the network delivers messages from one point to another in the order in which they were transmitted from the source, and that all the prefetches from a given processor in this loop are directed to the same source, then we can still simply read from the head of the prefetch buffer every time. The Cray T3D multiprocessor and the DEC Alpha processor it uses provide this form of prefetching.

Even if data cannot be prefetched early enough that they have arrived by the time the actual reference is made, prefetching may have several benefits. First, if the actual reference finds that the address it is accessing has an outstanding prefetch associated with it, then it can simply wait for the remaining time until the prefetched data returns. Also, depending on the number of prefetches allowed to be outstanding at a time, prefetches can be issued back to back to overlap their latencies. This can provide the benefits of pipelined data movement, although the pipeline rate may be limited by the overhead of issuing prefetches.

11.7.2 Cache-coherent Shared Address Space

In fact, precommunication is much more interesting in a cache-coherent shared address space: Shared nonlocal data may now be precommunicated directly into a processor's cache rather than a special buffer, and precommunication interacts with the cache coherence protocol. We shall therefore discuss the techniques in more detail in this context.

Consider an invalidation-based coherence protocol. A read miss fetches the data locally from wherever it is. A write that generates a read-exclusive fetches both data and ownership (by informing the home, perhaps invalidating other caches, and receiving acknowledgments), and a write that generates an upgrade fetches only ownership. All of these "fetches" have latency associated with them, so all are candidates for prefetching (we can prefetch data or ownership or both).

Update-based coherence protocols generate sender-initiated communication, and like other sender-initiated communication techniques they provide a form of precommunication from the viewpoint of the destinations of the updates. While update protocols are not very prevalent, techniques to selectively update copies can be used for sender-initiated precommunication even with an underlying invalidation-based protocol. One possibility is to insert software instructions that generate updates only on certain writes, to contain unnecessary traffic due to useless updates and hybrid update-invalidate. Some such techniques will be discussed later.

Prefetching Concepts

There are two broad categories of prefetching applicable to multiprocessors and uniprocessors: hardware-controlled and software-controlled prefetching. In hardware-controlled prefetching, no special instructions are added to the code: special hardware is used to try to predict future accesses from observed behavior, and prefetch data based on these predictions. In software-controlled prefetching, the decisions of what and when to prefetch are made by the programmer or compiler (hopefully the compiler!) by static analysis of the code, and prefetch are instructions inserted in the code based on this analysis. Tradeoffs between hardware- and software-controlled prefetching will be discussed later in this section. Prefetching can also be combined with block data transfer in *block prefetch* (receiver-initiated prefetch of a large block of data) and block put (sender-initiated) techniques.Let us begin by discussing some important concepts in prefetching in a cache-coherent shared address space.

In a multiprocessor, a key issue that dictates how early a prefetch can be issued in both software- and hardware-controlled prefetching in a multiprocessor is whether prefetches are *binding* or *non-binding*. A binding prefetch means that the value of the prefetched datum is bound at the time of the prefetch; i.e. when the process later reads the variable through a regular read, it will see the value that the variable had when it was prefetched, even if the value has been modified by another processor and become visible to the reader's cache in between the prefetch and the actual read. The prefetch we discussed in the non-cache-coherent case (prefetching into a prefetch buffer, see Figure 11-19) is typically a binding prefetch, as is prefetching directly into processor registers. A non-binding prefetch means that the value brought by a prefetch instruction remains subject to modification or invalidation until the actual operation that needs the data is executed. For example, in a cache-coherent system, the prefetched data are still visible to the coherence protocol; prefetched data are brought into the cache rather than into a register or prefetch buffer (which are typically not under control of the coherence protocol), and a modification by another processor that occurs between the time of the prefetch and the time of the use will update or invalidate the prefetched block according to the protocol. This means that non-binding prefetches can be issued at any time in a parallel program without affecting program semantics. Binding prefetches, on the other hand, affect program semantics, so we have to be careful about when we issue them. Since non-binding prefetches can to be generated as early as desired, they have performance advantages as well.

The other important concepts have to do with the questions of *what* data to prefetch, and *when* to initiate prefetches. Determining what to prefetch is called *analysis*, and determine when to initiate prefetches is called *scheduling*. The concepts that we need to understand for these two aspects of prefetching are those of prefetches being *possible*, obtaining good *coverage*, being *unnecessary*, and being *effective* [Mow94].

Prefetching a given reference is considered *possible* only if the address of the reference can be determined ahead of time. For example, if the address can be computed only just before the word is referenced, then it may not be possible to prefetch. This is an important consideration for applications with irregular and dynamic data structures implemented using pointers, such as linked lists and trees.

The coverage of a prefetching technique is the percentage of the original cache misses (without any prefetching) that the technique is able to prefetch. "Able to prefetch" can be defined in different ways; for example, able to prefetch early enough that the actual reference hits and all the

latency is hidden, or able to issue a prefetch earlier than just before the reference itself so that at least a part of the latency is hidden.

Achieving high coverage is not the only goal, however. In addition to not issuing prefetches for data that will not be accessed by the processor, we should not issue prefetches for data that are already in the cache to which data are prefetched, since these prefetches will just consume overhead and cache access bandwidth (interfering with regular accesses) without doing anything useful. More is not necessarily better for prefetching. Both these types of prefetches are called *unnecessary* prefetches. Avoiding them requires that we analyze the data locality in the application's access patterns and how it interacts with the cache size and organization, so that prefetches are issued only for data that are not likely to be present in the cache.

Finally, timing and luck play important roles in prefetching. A prefetch may be possible and not unnecessary, but it may be initiated too late to hide the latency from the actual reference. Or it may be initiated too early, so it arrives in the cache but is then either replaced (due to capacity or mapping conflicts) or invalidated before the actual reference. Thus, a prefetch should be *effective*: early enough to hide the latency, and late enough so the chances of replacement or invalidation are small.

As important as what and when to prefetch, then, are the questions of what not to prefetch and when not to issue prefetches. The goal of prefetching analysis is to maximize coverage while minimizing unnecessary prefetches, and the goal of scheduling to maximize effectiveness. Let us now consider hardware-controlled and software-controlled prefetching in some detail, and see how successfully they address these important aspects.

Hardware-controlled Prefetching

The goal in hardware-controlled prefetching is to provide hardware that can detect patterns in data accesses at runtime. Hardware-controlled prefetching assumes that accesses in the near future will follow these detected patterns. Under this assumption, the cache blocks containing those data can be prefetched and brought into the processor cache so the later accesses may hit in the cache. The following discussion assumes nonbinding prefetches.

Both analysis and scheduling are the responsibility of hardware, with no special software support, and both are performed dynamically as the program executes. Analysis and scheduling are also very closely coupled, since the prefetch for a cache block is initiated as soon as it is determined that the block should be prefetched: It is difficult for hardware to make separate decisions about these. Besides coverage, effectiveness and minimizing unnecessary prefetches, the hardware has two other important goals:

- it should be simple, inexpensive, and not take up too much area
- it should not be in the critical path of the processor cycle time as far as possible

Many simple hardware prefetching schemes have been proposed. At the most basic level, the use of long cache blocks itself is a form of hardware prefetching, exploited well by programs with good spatial locality. Essentially no analysis is used to restrict unnecessary prefetches, the coverage depends on the degree of spatial locality in the program, and the effectiveness depends on how much time elapses between when the processor access the first word (which issues the cache miss) and when it accesses the other words on the block. For example, if a process simply traverses a large array with unit stride, the coverage of the prefetching with long cache blocks

will be quite good (75% for a cache block of four words) and there will be no unnecessary prefetches, but the effectiveness is likely to not be great since the prefetches are issued too late. Extending the idea of long cache blocks or blocks are one-block lookahead or OBL schemes, in which a reference to cache block i may trigger a prefetch of block $i+1$. There are several variants for the analysis and scheduling that may be used in this technique; for example, prefetching $i+1$ whenever a reference to i is detected, only if a cache miss to i is detected, or when i is referenced for the first time after it is prefetched. Extensions may include prefetching several blocks ahead (e.g. blocks $i+1$, $i+2$, and $i+3$) instead of just one [DDS95], an adaptation of the idea of *stream buffers* in uniprocessors where several subsequent blocks are prefetched into a separate buffer rather than the cache when a miss occurs [Jou90]. Such techniques are useful when accesses are mostly unit-stride (the stride is the gap between data addresses accessed by consecutive memory operations).

The next step is to detect and prefetch accesses with non-unit or large stride. A simple way to do this is to keep the address of the previously accessed data item for a given instruction (i.e a given program counter value) in a history table indexed by program counter (PC). When the same instruction is issued again (e.g. in the next iteration of a loop) if it is found in the table the stride is computed as the difference between the current data address and the one in the history table for that instruction, and a prefetch issued for the data address ahead of the current one by that stride [FuP91,FPJ92]. The history table is thus managed much like a branch history table. This scheme is likely to work well when the stride of access is fixed but is not one; however, most other prefetching schemes that we shall discuss, hardware or software, are likely to work well in this case.

The schemes so far find ways to detect simple regular patterns, but do not guard against unnecessary prefetches when references do not follow a regular pattern. For example, if the same stride is not maintained between three successive accesses by the same instruction, then the previous scheme will prefetch useless data that the processor does not need. The traffic generated by these unnecessary prefetches can be very detrimental to performance, particularly in multiprocessors, since it consumes valuable resources thus causing contention and getting in the way of useful regular accesses.

In more sophisticated hardware prefetching schemes, the history table stores not only the data address accessed the last time by an instruction but also the stride between the previous two addresses accessed by it [BaC91,ChB92]. If the next data address accessed by that instruction is separated from the previous address by the same stride, then a regular-stride pattern is detected and a prefetch may be issued. If not, then a break in the pattern is detected and a prefetch is not issued, thus reducing unnecessary prefetches. In addition to the address and stride, the table entry also contains some state bits that keep track of whether the accesses by this instruction have recently been in a regular-stride pattern, have been in an irregular pattern, or are transitioning into or out of a regular-stride pattern. A set of simple rules is used to determine based on the stride match and the current state whether to potentially issue a prefetch or not. If the result is to potentially prefetch, the cache is looked up to see if the block is already there, and if not then the prefetch is issued.

While this scheme improves the analysis, it does not yet do a great job of scheduling prefetches. In a loop, it will prefetch only one loop iteration ahead. If the amount of work to do in a loop iteration is small compared to the latency that needs to be hidden, this will not be sufficient to tolerate the latency. The prefetched data will arrive too late and the processor will have to stall. On the other hand, if the work in a loop iteration is too much, the prefetched data will arrive too early

and may replace other useful blocks before being used. The goal of scheduling is to achieve just-in-time prefetching, that is, to have a prefetch issued about *l* cycles before the instruction that needs the data, where *l* is the latency we want to hide, so that a prefetched datum arrives just before the actual instruction that needs it is executed and the prefetch is likely to be effective (see Section 11.7.1). This means that the prefetch should be issued $\left\lceil \frac{l}{b} \right\rceil$ loop iterations (or times the instruction is encountered) ahead of the actual instruction that needs the data, where *b* is the predicted execution time of an iteration of the loop body.

One way to implement such scheduling in hardware is to use a "lookahead program counter" (LA-PC) that tries to remain *l* cycles ahead of the actual current PC, and that is used to access the history table and generate prefetches instead of (but in conjunction with) the actual PC. The LA-PC starts out a single cycle ahead of the regular PC, but keeps being incremented even when the processor (and PC) stalls on a cache miss, thus letting it get ahead of the PC. The LA-PC also looks up the branch history table, so the branch prediction mechanism can be used to modify it when necessary and try to keep it on the right track. When a mispredicted branch is detected, the LA-PC is set back to being equal to PC+1. A limit register controls how much farther the LA-PC can get ahead of the PC. The LA-PC stalls when this limit is exceeded, or when the buffer of outstanding references is full (i.e. it cannot issue any prefetches).

Both the LA-PC and the PC look up the prefetch history table every cycle (they are likely to access different entries). The lookup by the PC updates the "previous address" and the state fields in accordance with the rules, but does not generate prefetches. A new "times" field is added to the history table entries, which keeps track of the number of iterations (or encounters of the same instruction) that the LA-PC is ahead of the PC. The lookup by the LA-PC increments this field for the entry that it hits (if any), and generates an address for potential prefetching according to the rules. The address generated is the "times" field times the stride stored in the entry, plus the previous address field. The times field is decremented when the PC encounters that instruction in its lookup of the prefetch history table. More details on these hardware schemes can be found in [ChB92, Sin97].

Prefetches in these hardware schemes are treated as hints, since they are nonbinding, so actual cache misses get priority over them. Also, if a prefetch raises an exception (for example, a page fault or other violation), the prefetch is simply dropped rather than handling the exception. More elaborate hardware-controlled prefetching schemes have been proposed to try to prefetch references with irregular stride, requiring even more hardware support [ZhT95]. However, even the techniques described above have not found their way into microprocessors or multiprocessors. Instead, several microprocessors are preferring to provide prefetch instructions for use by software-controlled prefetching schemes. Let us examine software-controlled prefetching, before we examine its relative advantages and disadvantages with respect to hardware-controlled schemes.

Software-controlled Prefetching

In software-controlled prefetching, the analysis of what to prefetch and the scheduling of when to issue prefetches are not done dynamically by hardware, but rather statically by software. The compiler (or programmer) inserts special prefetch instructions into the code at points where it deems appropriate. As we saw in Figure 11-19, this may require restructuring the loops in the program to some extent as well. The hardware support needed is the provision of these non-blocking instructions, a cache that allows multiple outstanding accesses, and some mechanism

for keeping track of outstanding accesses (this is often combined with the write buffer in the form of an outstanding transactions buffer). The latter two mechanisms are in fact required for all forms of latency tolerance in a read-write based system.

Let us first consider software-controlled prefetching from the viewpoint of a given processor trying to hide latency in its reference stream, without complications due to interactions with other processors. This problem is equivalent to prefetching in uniprocessors, except with a wider range of latencies. Then, we shall discuss the complications introduced by multiprocessing.

Prefetching With a Single Processor

Consider a simple loop, such as our example from Figure 11-1. A simple approach would be to always issue a prefetch instruction one iteration ahead on array references within loops. This would lead to a software pipeline like the one in Figure 11-19 on page 794, with two differences: The data are brought into the cache rather than into a prefetch buffer, and the later load of the data will be from the address of the data rather than from the prefetch buffer (i.e. `read(A[i])` and `use(A[i])`). This can easily be extended to approximate "just-in-time" prefetching above by issuing prefetches multiple iterations ahead as discussed above. You will be asked to write the code for just-in-time prefetching for a loop like this in Exercise 11.9.

To minimize unnecessary prefetches, it is important to analyze and predict the temporal and spatial locality in the program as well as trying to predict the addresses of future references. For example, blocked matrix factorization reuses the data in the current block many times in the cache, so it does not make sense to prefetch references to the block even if they can be predicted early. In the software case, unnecessary prefetches have an additional disadvantage beyond useless traffic and cache bandwidth (even when traffic is not generated): They introduce useless prefetch instructions in the code, which add execution overhead. The prefetch instructions are often placed within conditional expressions, and with irregular data structures extra instructions are often needed to compute the address to be prefetched, both of which increase the instruction overhead.

How easy is it to analyze what references to prefetch in software? In particular, can a compiler do it, or must it be left to the programmer? The answer depends on the program's reference patterns. References are most predictable when they correspond to traversing an array in some regular way. For example, a simple case to predict is when an array is referenced inside a loop or loop nest, and the array index is an affine function of the loop indices in the loop nest; i.e. a linear combination of loop index values. The following code shows an example:

```
for i <- 1 to n
    for j <- 1 to n
        sum = sum + A[3i+5j+7];
    end for
end for.
```

Given knowledge of the amount of latency we wish to hide for a reference, we can try to issue the prefetch that many iterations ahead in the relevant loop. Also, the amount of array data accessed and the spatial locality in the traversal are easy to predict, which makes locality analysis easy. The major complication in analyzing data locality is predicting cache misses due to mapping conflicts.

A slightly more difficult class of references to analyze is indirect array references. e.g.

```
for i <- 1 to n
    sum = sum + A[index[i]];
end for.
```

While we can easily predict the values of `i` and hence the elements of the `index` array that will be accessed, we cannot predict the value in `index[i]` and hence the elements of `A` that we shall access. What we must do in order to predict the accesses to `A` is to first prefetch `index[i]` well enough in advance, and then use the prefetched value to determine the element of array `A` to prefetch. The latter requires additional instructions to be inserted. Consider scheduling. If the number of iterations that we would normally prefetch ahead is k, we should prefetch `index[i]` $2k$ iterations ahead so it returns k iterations before we need `A[index[i]]`, at which point we can use the value of `index[i]` to prefetch `A[index[i]]` (see Exercise a). Analyzing temporal and spatial locality in these cases is more difficult than predicting addresses. It is impossible to perform accurate analysis statically, since we do not know ahead of time how many different locations of `A` will be accessed (different entries in the index array may have the same value) or what the spatial relationships among the references to A are. Our choices are therefore to either prefetch all references `A[index[i]]` or none at all, to obtain profile information about access patterns gathered at runtime and use it to make decisions, or to use higher-level programmer knowledge.

Compiler technology has advanced to the point where it can handle the above types of array references in loops quite well [Mow94], within the constraints described above. Locality analysis [WoL91] is first used to predict when array references are expected to miss in a given cache (typically the first-level cache). This results in a prefetch predicate, which can be thought of as a conditional expression inside which a prefetch should be issued for a given iteration. Scheduling based on latency is then used to decide how many iterations ahead to issue the prefetches. Since the compiler may not be able to determine which level of the extended memory hierarchy the miss will be satisfied in, it may be conservative and assume the worst-case latency.

Predicting conflict misses is particularly difficult. Locality analysis may tell us that a word should still be in the cache so a prefetch should not be issued, but the word may be replaced due to conflict misses and therefore would benefit from a prefetch. A possible approach is to assume that a small-associativity cache of C bytes effectively behaves like a fully-associative cache that is smaller by some factor, but this is not reliable. Multiprogramming also throws off the predictability of misses across process switches, since one process might pollute the cache state of another that is prefetching, although the time-scales are such that locality analysis often doesn't assume state to stay in the cache that long anyway. Despite these problems, limited experiments have shown quite good potential for success with compiler-generated prefetching when most of the cache misses are to affine or indirect array references in loops [Mow94], as we shall see in Section 11.7.3. These include programs on regular grids or dense matrices, as well as sparse matrix computations (in which the sparse matrices use indirect array references, but most of the data are often stored in dense form anyway for efficiency). These experiments are performed through simulation, since real machines are only just beginning to provide effective underlying hardware support for software-controlled prefetching.

Accesses that are truly difficult for a compiler to predict are those that involve pointers or recursive data structures (such as linked lists and trees). Unlike array indexing, traversing these data structures requires dereferencing pointers along the way; the address contained in the pointer

within a list or tree element is not known until that element is reached in the traversal, so it cannot be easily prefetched. Predicting locality for such data structures is also very difficult, since their traversals are usually highly irregular in memory. Prefetching in these cases currently must be done by the programmer, exploiting higher-level semantic knowledge of the program and its data structures as shown in the following example. Compiler analysis for prefetching recursive data structures is currently the subject of research [LuM96], and will also be helped by progress in alias analysis for pointers.

Example 11-2   Consider the tree traversal to compute the force on a particle in the Barnes-Hut application described in Chapter 3. The traversal is repeated for each particle assigned to the process, and consecutive particles reuse much of the tree data, which is likely to stay in the cache across particles. How should prefetches be inserted in this tree traversal code?

Answer   The traversal of the octree proceeds in depth first manner. However, if it is determined that a tree cell needs to be opened then all its eight children will be examined as long they are present in the tree. Thus, we can insert prefetches for all the children of a cell as soon as we determine that the cell will be opened (or we can speculatively issue prefetches as soon as we touch a cell and can therefore dereference the pointers to its children). Since we expect the working set to fit in the cache (which a compiler is highly unlikely to be able to determine), to avoid unnecessary prefetches it is likely that we should need to prefetch a cell only the first time that we access it (i.e. for the first particle that accesses it), not for subsequent particles. Cache conflicts may occur which cause unpredictable misses, but there is likely little we can do about that statically. Note that we need to do some work to generate prefetches (determine if this is the first time we are visiting the cell, access and dereference the child pointers etc.), so the overhead of a prefetch is likely to be several instructions. If the overhead is incurred a lot more often than successful prefetches are generated, it may overcome the benefits of prefetching. One problem with this scheme is that we may not be prefetching early enough when memory latencies are high. Since we prefetch all the children at a time, in most cases the depth first work done for the first child (or two) should be enough to hide the latency of the rest of the children, but this may not be the case. The only way to improve this is to speculatively prefetch multiple levels down the tree when we encounter a cell, hoping that we will touch all the cells we prefetch and they will still be in the cache when we reach them. Other applications that use linked lists in unstructured ways may be even more difficult for a compiler or even programmer to prefetch successfully.

Finally, sometimes a compiler may fail to determine what to prefetch because other aspects of the compiler technology are not up to their task. For example, the key references in a loop may be within a procedure call, and the compiler's interprocedural analysis may not be sophisticated enough to detect that the references in the procedure are indeed affine functions of the indices of the loop indices. If the called procedure is in a different file, the compiler would have to perform interprocedural analysis or inlining across files. These sorts of situations are easy for the programmer, who has higher-level semantic knowledge of what is being accessed.

Additional Complications in Multiprocessors

Two additional issues that we must consider when prefetching in parallel programs are prefetching communication misses, and prefetching with ownership. Both arise from the fact that other processors might also be accessing and modifying the data that a process references.

In an invalidation-based cache-coherent multiprocessor, data may be removed from a processor's cache—and misses therefore incurred— not only due to replacements but also due to invalidations. A similar locality analysis is needed for prefetching in light of communication misses: We should not prefetch data so early that they might be invalidated in the cache before they are used, and we should ideally recognize when data might have been invalidated so that we can prefetch them again before actually using them. Of course, because we are using nonbinding prefetching, these are all performance issues rather than correctness issues. We could insert prefetches into a process's reference stream as if there were not other processes in the program, and the program would still run correctly.

It is difficult for the compiler to predict incoming invalidations and perform this analysis, because the communication in the application cannot be easily deduced from an explicitly parallel program in a shared address space. The one case where the compiler has a good chance is when the compiler itself parallelized the program, and therefore knows all the necessary interactions of when different processes read and modify data. But even then, dynamic task assignment and false sharing of data greatly complicate the analysis.

A programmer has the semantic information about interprocess communication, so it is easier for the programmer to insert and schedule prefetches as necessary in the presence of invalidations. The one kind of information that a compiler does have is that conveyed by explicit synchronization statements in the parallel program. Since synchronization usually implies that data are being shared (for example, in a "properly labeled" program, see Chapter 9, the modification of a datum by one process and its use by another process is separated by a synchronization), the compiler analysis can assume that communication is taking place and that all the shared data in the cache have been invalidated whenever it sees a synchronization event. Of course, this is very conservative, and it may lead to a lot of unnecessary prefetches especially when synchronization is frequent and little data is actually invalidated between synchronization events. It would be nice if a synchronization event conveyed some information about which data might be modified, but this is usually not the case.

As a second enhancement, since a processor often wants to fetch a cache block with exclusive ownership (i.e. in exclusive mode, in preparation for a write), or to simply obtain exclusivity by invalidating other cached copies, it makes sense to prefetch data with exclusivity or simply to prefetch exclusivity. This can have two benefits when used judiciously. First, it reduces the latency of the actual write operations that follow, since the write does not have to invalidate other blocks and wait to obtain ownership (that was already done by the prefetch). Whether or not this in itself has an impact on performance depends on whether write latency is already hidden by other methods like using a relaxed consistency model. The second advantage is in the common case where a process first reads a variable and then shortly thereafter writes it. Prefetching with ownership even before the read in this case halves the traffic, as seen in Figure 11-20, and hence improve the performance of other references as well by reducing contention and bandwidth needs. Quantitative benefits of prefetching in exclusive more are discussed in [Mow94].
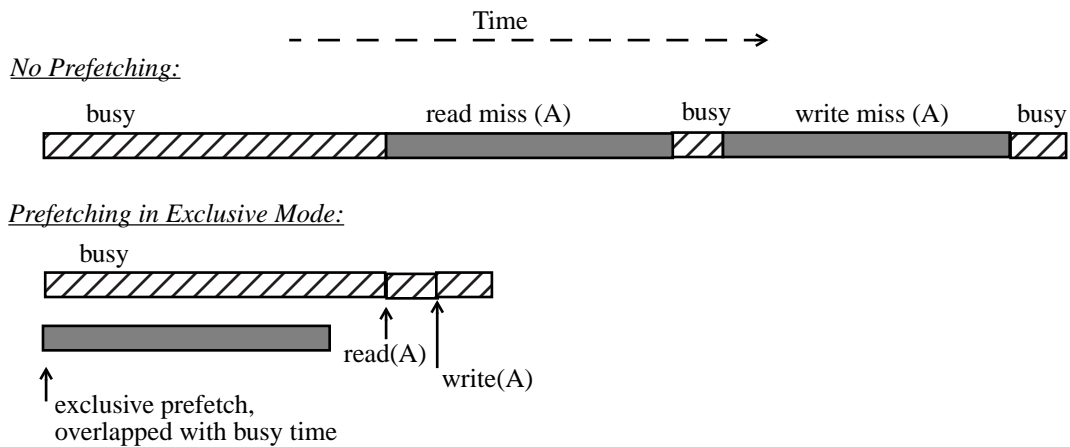
Figure 11-20 Benefits from prefetching with ownership.

Suppose the latest copy of A is not available locally to begin with, but is present in other caches. Normal hardware cache coherence would fetch the corresponding block in shared state for the read, and then communicate again to obtain ownership upon the write. With prefetching, if we recognize this read-write pattern we can issue a single prefetch with ownership before the read itself, and not have to incur any further communication at the write. By the time the write occurs, the block is already there in exclusive state.

Hardware-controlled versus Software-controlled Prefetching

Having seen how hardware-controlled and software-controlled prefetching work, let us discuss some of their relative advantages and disadvantages. The most important advantages of hardware-controlled prefetching are (i) it does not require any software support from the programmer or compiler, (ii) it does not require recompiling code (which may be very important in practice when the source code is not available), and (iii) it does not incur any instruction overhead or code expansion since no extra instructions are issued by the processor. On the other hand, its most obvious disadvantages are that it requires substantial hardware overhead and the prefetching "algorithms" are hard-wired into the machine. However, there are many other differences and tradeoffs, having to do with coverage, minimizing unnecessary prefetches and maximizing effectiveness. Let us examine these tradeoffs, focusing on compiler-generated rather than programmer-generated prefetches in the software case.

Coverage: The hardware and software schemes take very different approaches to analyzing what to prefetch. Software schemes can examine all the references in the code, while hardware simply observes a window of dynamic access patterns and predicts future references based on current patterns. In this sense, software schemes have greater potential for achieving coverage of complex access patterns. Software may be limited in its analysis (e.g. by the limitations of pointer and interprocedural analysis) while hardware may be limited by the cost of maintaining sophisticated history and the accuracy of needed techniques like branch prediction. The compiler (or even programmer) also cannot react to some forms of truly dynamic information that are not predicted by its static analysis, such as the occurrence of replacements due to cache conflicts, while hardware can. Progress is being made in improving the coverage of both approaches in prefetching more types of access patterns [ZhT95, LuM96], but the costs in the hardware case appear high. It is possible to use some runtime feedback from hardware (for example conflict miss pat-

terns) to improve software prefetching coverage, but there has not been much progress in this direction.

Reducing unnecessary prefetches: Hardware prefetching is driven by increasing coverage, and does not perform any locality analysis to reduce unnecessary prefetches. The lack of locality or reuse analysis may generate prefetches for data that are already in the cache, and the limitations of the analysis may generate prefetches to data that the processor does not access. As we have seen, these prefetches waste cache access bandwidth even if they find the data in the cache and are not propagated past it. If they do actually prefetch the data, they waste memory or network bandwidth as well, and can displace useful data from the processor's working set in the cache. Especially on a bus-based machine, wasting too much interconnect bandwidth on prefetches has at least the potential to saturate the bus and reduce instead of enhancing performance [Tue93].

Maximizing effectiveness: In software prefetching, scheduling is based on prediction. However, it is often difficult to predict how long a prefetch will take to complete, for example where in the extended memory hierarchy it will be satisfied and how much contention it will encounter. Hardware can in theory automatically adapt to how far ahead to prefetch at runtime, since it lets the lookahead-PC get only as far ahead as it needs to. But hiding long latencies becomes difficult due to branch prediction, and every mispredicted branch causes the lookahead-PC to be reset leading to a few ineffective prefetches until it gets far enough ahead of the PC again. Thus, both the software and hardware schemes have potential problems with effectiveness or just-in-time prefetching.

Hardware prefetching is used in dynamically scheduled microprocessors to prefetch data for operations that are waiting in the reorder buffer but cannot yet be issued, as discussed earlier. However, in that case, hardware does not have to detect patterns and analyze what to prefetch. So far, the on-chip hardware support needed for more general hardware analysis and prefetching of non-unit stride accesses has not been considered worthwhile. On the other hand, microprocessors are increasingly providing prefetching instructions to be used by software (even in uniprocessor systems). Compiler technology is progressing as well. We shall therefore focus on software-controlled prefetching in our quantitative evaluations of prefetching in the next section. After this evaluation, we shall examine more closely some of the architectural support needed by and tradeoffs encountered in the implementation of software-controlled prefetching.

Policy Issues and Tradeoffs

Two major issues that a prefetching scheme must take a position on are when to drop prefetches that are issued, i.e. not propagate them any further, and where to place the prefetched data.

Since they are hints, prefetches can be dropped by hardware when their costs seem to outweigh their benefits. Two interesting questions are whether to drop a prefetch when the prefetch instruction incurs a TLB miss in address translation, and whether to drop it when the buffer of outstanding prefetches or transactions is already full. TLB misses are expensive, but if they would occur anyway at the actual reference to the address then it is probably worthwhile to incur the latency at prefetch time rather than at reference time. However, the prefetch may be to an invalid address and would not be accessed later, but this is not known until the translation is complete. One possible solution is to guess: to have two flavors of prefetches, one that gets dropped on TLB misses and the other that does not, and insert the appropriate one based on the expected nature of the TLB miss if any. Whether dropping prefetches when the buffer that tracks outstanding accesses is

full is a good idea depends on the extent to which we expect contention for the buffer with ordinary reads and writes, and the extent to which we expect prefetches to be useless. If both these extents are high, we would be inclined to drop prefetches in this case.

Prefetched data can be placed in the primary cache, in the secondary cache, or in a separate prefetch buffer. A separate buffer has the advantage that prefetched data do not conflict with useful data in a cache, but it requires additional hardware and must be kept coherent if prefetches are to be non-binding. While ordinary misses replicate data at every level of the cache hierarchy, on prefetches we may not want to bring the data very close to the processor since they may replace useful data in the small cache and may compete with processor loads and stores for cache bandwidth. However, the closer to the processor we prefetch, the more the latency we hide if the prefetches are successful. The correct answer depends on the size of the cache and the important working sets, the latencies to obtain data from different levels, and the time for which the primary cache is tied up when prefetched data are being put in it, among other factors (see Exercise f). Results in the literature suggest that the benefits of prefetching into the primary cache usually outweigh the drawbacks [Mow94].

Sender-initiated Precommunication

In addition to hardware update-based protocols, support for software-controlled "update", "deliver", or "producer prefetch" instructions has been implemented and proposed. An example is the "poststore" instruction in the KSR-1 multiprocessor from Kendall Square Research, which pushes the contents of the whole cache block into the caches that currently contain a (presumably old) copy of the block; other instructions push only the specified word into other caches that contain a copy of the block. A reasonable place to insert these update instructions is at the last write to a shared cache block before a release synchronization operation, since it is likely that those data will be needed by consumers. The destination nodes of the updates are the sharers in the directory entry. As with update protocols, the assumption is that past sharing patterns are a good predictor of future behavior. (Alternatively, the destinations may be specified in software by the instruction itself, or the data my be pushed only to the home main memory rather than other caches, i.e. a write-through rather than an update, which hides some but not all of the latency from the destination processors.) These software-based update techniques have some of the same problems as hardware update protocols, but to a lesser extent since not every write generates a bus transaction. As with update protocols, competitive hybrid schemes are also possible [Oha96, GrS96].

Compared to prefetching, the software-controlled sender-initiated communication has the advantage that communication happens just when the data are produced. However, it has several disadvantages. First, the data may be communicated too early, and may be replaced from the consumer's cache before use, particularly if they are placed in the primary cache. Second, unlike prefetching this scheme precommunicates only communication (coherence) misses, not capacity or conflict misses. Third, while a consumer knows what data it will reference, a producer may deliver unnecessary data into processor caches if past sharing patterns are not a perfect predictor of future patterns, which they often aren't, or may even deliver the same data value multiple times. Fourth, a prefetch checks the cache and is dropped if the data are found in the cache, reducing unnecessary network traffic. The software update performs no such checks and can increase traffic and contention, though it reduces traffic when it is successful since it deposits the data in the right places without requiring multiple protocol transactions. And finally, since the receiver does not control how many precommunicated messages it receives, buffer overflow and

the resulting complications at the receiver may easily occur. The wisdom from simulation results so far is that prefetching schemes work better than deliver or update schemes for most applications [Oha96], though the two can complement each other if both are provided [AHA+97].

Finally, both prefetch and deliver can be extended with the capability to transfer larger blocks of data (e.g. multiple cache blocks, a whole object, or an arbitrarily defined range of addresses) rather than a single cache block. These are called block prefetch and block put mechanisms (block put differs from block transfer in that the data are deposited in the cache and not in main memory). The issues here are similar to those encountered by prefetch and software update except for differences due to their size. For example, it may not be a good idea to prefetch or deliver a large block to the primary cache.

Since software-controlled prefetching is generally more popular than hardware-controlled prefetching or sender-initiated precommunication, let us briefly examine some results from the literature that illustrate the performance improvements it can achieve.

11.7.3 Performance Benefits

Performance results from prefetching so far have mostly been examined through simulation. To illustrate the potential, let us examine results from hand-inserted prefetches in some of the example applications used in this book [WSH95]. Hand-inserted prefetches are used since they can be more aggressive then the best available compiler algorithms. Results from state-of-the-art compiler algorithms will also be discussed.

Benefits with Single-issue, Statically Scheduled Processors

Section 11.5.3 illustrated the performance benefits for block transfer on the FFT and Ocean applications for a particular simulated platform. Let us first look at how prefetching performs for the same programs and platform. This experiment focuses on prefetching only remote accesses (misses that cause communication). Figure 11-21 shows the results for FFT. Comparing the prefetching results with the original non-prefetched results, we see in the first graph that prefetching remote data helps performance substantially in this predictable application. As with block transfer, the benefits are less for large cache blocks than for small ones, since there is a lot of spatial locality in the applications so large cache blocks already achieve prefetching in themselves. The second graph compares the performance of block transfer with that of the prefetched version, and shows that the results are quite similar: prefetching is able to deliver most of the benefits of even the very aggressive block transfer in this application, as long as enough prefetches are allowed to be outstanding at a time (this experiment allows a total of 16 simultaneous outstanding operations from a processor). Figure 11-22 shows the same results for the Ocean application. Like block transfer, prefetching helps less here since less time is spent in communication and since not all of the prefetched data are useful (due to poor spatial locality on communicated data along column-oriented partition boundaries).

These results are only for prefetching remote accesses. Prefetching is often much more successful on local accesses (like uniprocessor prefetching). For example, in the nearest-neighbor grid computations in the Ocean application, with barriers between sweeps it is difficult to issue prefetches well enough ahead of time for boundary elements from a neighbor partition. This is another reason for the benefits from prefetching remote data not being very large. However, a process can very easily issue prefetches early enough for grid points within its assigned partition,
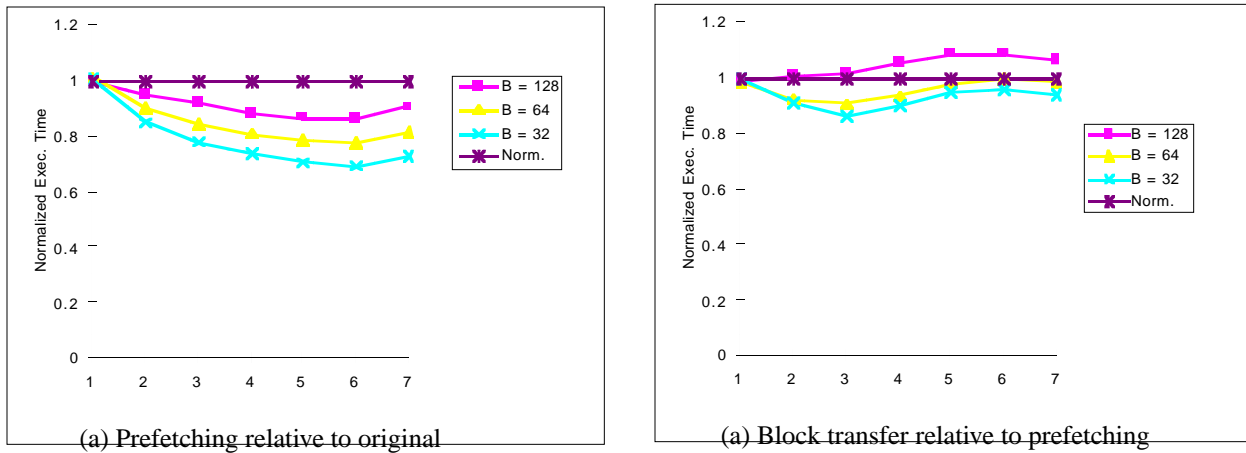
(a) Prefetching relative to original   (a) Block transfer relative to prefetching

Figure 11-21 Performance benefits of prefetching remote data in a Fast Fourier Transform.

The graph on the left shows the performance of the prefetched version relative to that of the original version. The graph can be interpreted just as described in Figure 11-13 on page 778: Each curve shows the execution time of the prefetched version relative to that of the non-prefetched version for the same cache block size (for each number of processors). Graph (b) shows the performance of the version with block transfer (but no prefetching), described earlier, relative to the version with prefetching (but no block transfer) rather than relative to the original version. It enables us to compare the benefits from block transfer with those from prefetching remote data.

(a) Prefetching relative to original   (a) Block transfer relative to prefetching
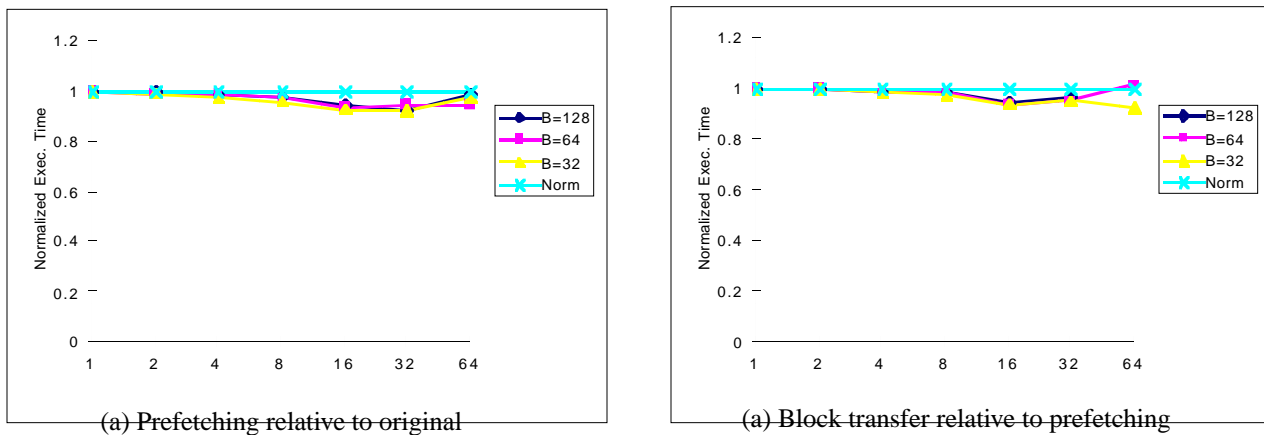
Figure 11-22 Performance benefits of prefetching remote data in Ocean.

The graphs are interpreted in exactly the same way as in Figure 11-21

which are not touched by any other process. Results from a state-of-the-art compiler algorithm show that the compiler can be quite successful in prefetching regular computations on dense arrays, where the access patterns are very predictable [Mow94]. These results, shown for two applications in Figure 11-23, include both local and remote accesses for 16-processor executions. The problems in these cases typically are being able to prefetch far enough ahead of time (for example when the misses occur at the beginning of a loop nest or just after a synchronization point), and being able to analyze and predict conflict misses.

(a) Ocean   (b) LU

Figure 11-23 Performance benefits from compiler-generated prefetching.

The results are for an older version of the Ocean simulation program (which partitions in chunks of complete rows and hence has a higher inherent communication to computation ratio) and for an unblocked and hence lower-performance dense LU factorization. There are three sets of b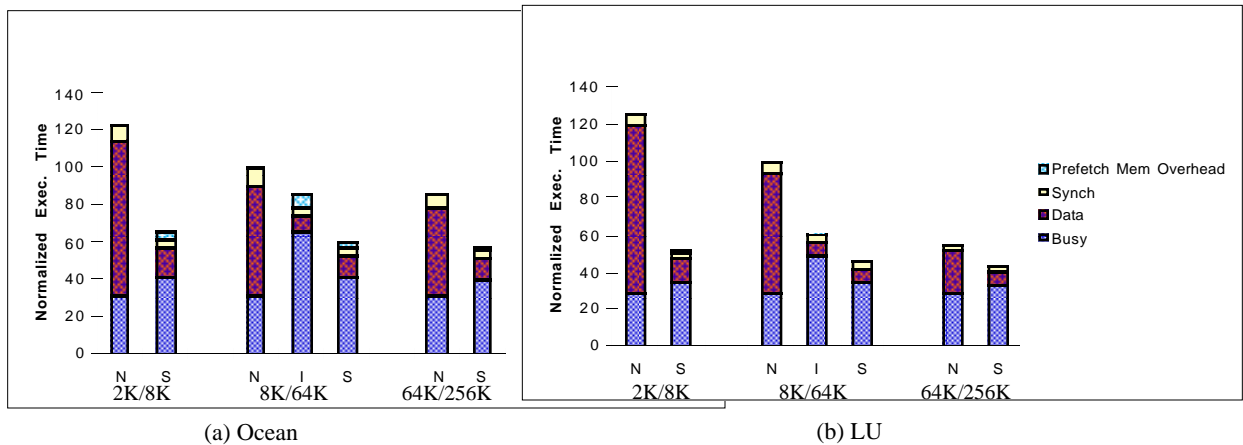ars, for different combinations of L1/L2 cache sizes. The bars for each combination are the execution times for no prefetching (N) and selective prefetching(S). For the intermediate combination (8K L1 cache and 64K L2 cache), results are also shown for the case where prefetches are issued indiscriminately (I), without locality analysis. All execution times are normalized to the time without prefetching for the 8K/64K cache size combination. The processor, memory, and communication architecture parameters are chosen to approximate those of the Stanford DASH multiprocessor, which is a relatively old multiprocessor, and can be found in [Mow94]. Latencies on modern systems are much larger relative to processor speed than on DASH. We can see that prefetching helps performance, and that the choice of cache sizes makes a substantial difference to the impact of prefetching. The increase in "busy" time with prefetching (especially indiscriminate prefetching) is due to the fact that prefetch instructions are included in busy time.

Some success has also been achieved on sparse array or matrix computations that use indirect addressing, but more irregular, pointer-based applications have not seen much success through compiler-generated prefetching. For example, the compiler algorithm is not able to successfully prefetch the data in the tree traversals in the Barnes-Hut application, both because temporal locality on these accesses is high but difficult to analyze, and because it is difficult to dereference the pointers and issue the prefetches well enough ahead of time. Another problem is that many applications with such irregular access patterns exhibit a high degree of temporal locality; this locality is very difficult to analyze, so even when the prefetching analysis is successful it can lead to a lot of unnecessary prefetches. Programmers can often do a better job in these cases, as discussed earlier, and runtime profile data may be useful to identify the data accesses that generate the most misses.

For the cases where prefetching is successful overall, compiler algorithms that use locality analysis to reduce unnecessary prefetch instructions have been found to reduce the number of prefetches issued substantially without losing much in coverage, and hence to perform much better (see Figure 11-24). Prefetching with ownership is found to hide write latency substantially in an architecture in which the processor implements sequential consistency by stalling on writes, but is less important when write latency is already being hidden through a relaxed memory consistency model. It does reduce traffic substantially in any case.

Quantitative evaluations in the literature [Mow94, ChB94] also show that as long as caches are reasonably large, cache intereference effects due to prefetching are negligible. They also illustrate that by being more selective, software prefetching indeed tends to induce less unnecessary
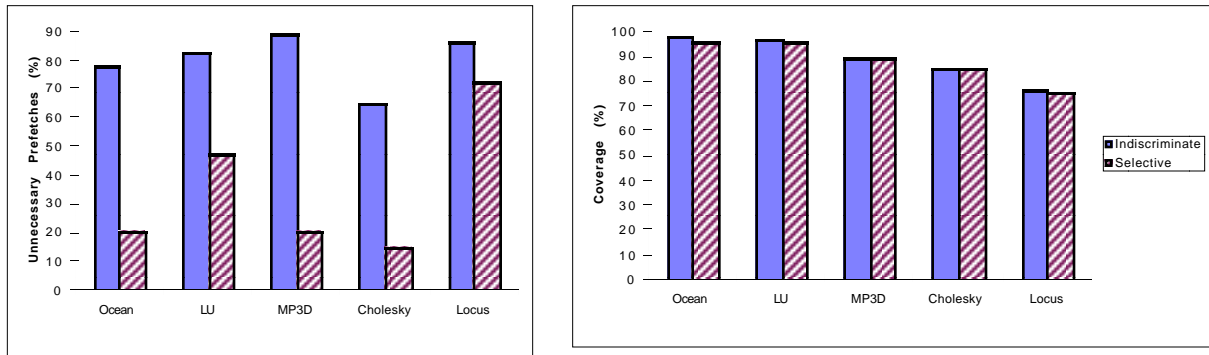
Figure 11-24 The benefits of selective prefetching through locality analysis.

The fraction of prefetches that are unnecessary is reduced substantially, while the coverage is not compromised. MP3D is an application for simulating rarefied hydrodynamics, Cholesky is a sparse matrix factorization kernel, and LocusRoute is a wire-routing application from VLSI CAD. MP3D and Cholesky use indirect array accesses, and LocusRoute uses pointers to implement linked lists so is more difficult for coverage.

extra traffic and cache conflict misses than hardware prefetching. However, the overhead due to extra instructions and associated address calculations can sometimes be substantial in software schemes.

Benefits with Multiple-issue, Dynamically Scheduled Processors

While the above results were obtained through simulations that assumed simple, statically scheduled processors, the effectiveness of software-controlled prefetching has also been measured (through simulation) on multiple-issue, dynamically scheduled processors [LuM96, BeF96a, BeF96b] and compared with the effectiveness on simple, statically scheduled processors [RPA+97]. Despite the latency tolerance already provided by dynamically scheduled processors (including some through hardware prefetching of operations in the reorder buffer, as discussed earlier), software-controlled prefetching is found to be effective in further reducing execution time. The reduction in memory stall time is somewhat smaller in dynamically scheduled processors than it is in statically scheduled processors. However, since memory stall time is a greater fraction of execution time in the former than in the latter (dynamically scheduled superscalar processors reduce instruction processing time much more effectively than they can reduce memory stall time), the percentage improvement in overall execution time due to prefetching is often comparable in the two cases.

There are two main reasons for prefetching being less effective in reducing memory stall time with dynamically scheduled superscalar processors than with simple processors. First, since dynamically scheduled superscalar processors increase the rate of instruction processing quite rapidly, there is not so much computation time to overlap with prefetches and prefetches often end up being late. The second reason is that dynamically scheduled processors tend to cause more resource contention for processor resources (outstanding request tables, functional units, tracking buffers, etc.) that are encountered by a memory operation even before the L1 cache. This is because they cause more memory operations to be outstanding at a time, and they do not block on read misses. Prefetching tends to further increase the contention for these resources, thus increasing the latency of non-prefetch accesses. Since this latency occurs before the L1 cache, it is not hidden effectively by prefetching. Resource contention of this sort is also the reason that

simply issuing prefetches earlier does not solve the late prefetches problem: Early prefetches not only are often wasted, but also tie up these processor resources for an even longer time since they tend to keep more prefetches outstanding at a time. The study in [RPA+97] was unable to improve performance significantly by varying how early prefetches are issued. One advantage of dynamically scheduled superscalar processors compared to single-issue statically scheduled processors from the viewpoint of prefetching is that the instruction overhead of prefetching is usually much smaller, since the prefetch instructions can occupy empty slots in the processor and are hence overlapped with other instructions.

Comparison with Relaxed Memory Consistency

Studies that compare prefetching with relaxed consistency models have found that on statically scheduled processors with blocking reads, the two techniques are quite complementary [GHG+91]. Relaxed models tend to greatly reduce write stall time but do not do much for read stall time in these blocking-read processors, while prefetching helps to reduce read stall time. Even after adding prefetching, however, there is a substantial difference in performance between sequential and relaxed consistency, since prefetching is not able to hide write latency as effectively as relaxed consistency. On dynamically scheduled processors with nonblocking reads, relaxed models are helpful in reducing read stall time as well as write stall time. Prefetching also helps reduce both, so it is interesting to examine whether starting from sequential consistency performance is helped more by prefetching or by using a relaxed consistency model. Even when all optimizations to improve the performance of sequential consistency are applied (like hardware prefetching, speculative reads, and write-buffering), it is found to be more advantageous to use a relaxed model without prefetching than to use a sequentially consistent model with prefetching on dynamically scheduled processors [RPA+97]. The reason is that although prefetching can help reduce read stall time somewhat better than relaxed consistency, it does not help sequential consistency hide write latency nearly as well as can be done with relaxed consistency.

11.7.4 Implementation Issues

In addition to extra bandwidth and lockup-free caches that allow multiple outstanding accesses (prefetches and ordinary misses), there are two types of architectural support needed for software-controlled prefetching: instruction set enhancements, and keeping track of prefetches. Let us look at each briefly.

Instruction Set Enhancements

Prefetch instructions differ from normal load instructions in several ways. First, there is no destination register with non-binding prefetches. Other differences are that prefetches are *non-blocking* and *non-excepting* instructions. The non-blocking property allows the processor to proceed past them. Since prefetches are hints and since no instructions depend on the data they return, unlike nonblocking loads they do not need support for dynamic scheduling, instruction lookahead and speculative execution to be effective. Prefetches do not generate exceptions when the address being prefetched is invalid; rather, they are simply ignored since they are only hints. This allows us to be aggressive in generating prefetches without analysis for exceptions (e.g. prefetching past array bounds in a software pipeline, or prefetching a data-dependent address before the dependence is resolved).

Prefetches can be implemented either as a special flavor of a load instruction, using some extra bits in the instruction, or using an unused opcode. Prefetches themselves can have several flavors, such as prefetching with or without ownership, prefetching multiple cache blocks rather than one, prefetching data in uncached mode rather than into the cache (in which case the prefetches are binding since the data are no longer visible to the coherence protocol), and flavors that determine whether prefetches should be dropped under certain circumstances. The issues involved in choosing an instruction format will be discussed in Exercise d.

Keeping Track of Outstanding Prefetches

To support multiple outstanding reads or writes, a processor must maintain state for each outstanding transaction: in the case of a write, the data written to must be merged with the rest of the cache block when it returns, while in the case of a read the register being load into must be kept track of and interlocks with this register handled. Since prefetches are just hints, it is not necessary to maintain processor state for outstanding prefetches, as long as the cache can receive prefetch responses from other nodes while the local processor might be issuing more requests. All we really need is lockup-free caches. Nonetheless, having a record of outstanding prefetches in the processor may be useful for performance reasons. First, it allows the processor to proceed past a prefetch even when the prefetch cannot be serviced immediately by the lockup-free cache, by buffering the prefetch request till cache can accept it. Second, it allows subsequent prefetches to the same cache block to be merged by simply dropping them. Finally, a later load miss to a cache block that has an outstanding prefetch can be merged with that prefetch and simply wait for it to return, thus partially hiding the latency of the miss. This is particularly important when the prefetch cannot be issued far enough before the actual miss: If there were no record of outstanding prefetches, the load miss would simply be issued and would incur its entire latency.

The outstanding prefetch buffer may simply be merged with an existing outstanding access buffer (such as the processor's write buffer), or separate structures may be maintained. The disadvantage of a combined outstanding transaction buffer is that prefetches may be delayed behind writes or vice versa; the advantage is that a single structure may be simpler to build and manage. In terms of performance, the difference between the two alternatives diminishes as more entries are used in the buffer(s), and may be significant for small buffers.

11.7.5 Summary

To summarize our discussion of precommunication in a cache-coherent shared address space, the most popular method so far is for microprocessors to provide support for prefetch instructions to be used by software-controlled prefetches, whether inserted by a compiler or a programmer. The same instructions are used for either uniprocessor or multiprocessor systems. In fact, prefetching turns out to be quite successful in competing with block data transfer even in cases where the latter technique works well, even though prefetching involves the processor on every cache block transfer. For general computations, prefetching has been found to be quite successful in hiding latency in predictable applications with relatively regular data access patterns, and successful compiler algorithms have been developed for this case. Programmer-inserted prefetching still tends to outperform compiler-generated prefetching, since the programmer has knowledge of access patterns across computations that enable earlier or better scheduling of prefetches. However, prefetching irregular computations, particularly those that use pointers heavily, has a long way to go. Hardware prefetching is used in limited forms as in dynamically scheduled processors; while it has important advantages in not requiring that programs be re-compiled, it's future

in microprocessors is not clear. Support for sender-initiated precommunication instructions is also not so popular as support for prefetching.

11.8 Multithreading in a Shared Address Space

Hardware-supported multithreading is perhaps the most versatile technique in terms of hiding different types of latency and in different circumstances. It has the following conceptual advantages over other approaches:

- it requires no special software analysis or support (other than having more explicit threads or processes in the parallel program than the number of processors),
- because it is invoked dynamically, it can handle unpredictable situations like cache conflicts etc. just as well as predictable ones.
- while the previous techniques are targeted at hiding memory access latency, multithreading can potentially hide the latency of any long-latency event just as easily, as long as the event can be detected at runtime. This includes synchronization and instruction latency as well.
- like prefetching, it does not change the memory consistency model since it does not reorder accesses within a thread.

Despite these potential advantages, multithreading is currently the least popular latency tolerating technique in commercial systems, for two reasons. First, it requires substantial changes to the microprocessor architecture. Second, its utility has so far not been adequately proven for uniprocessor or desktop systems, which constitute the vast majority of the marketplace. We shall see why in the course of this discussion. However, with latencies becoming increasingly longer relative to processor speeds, with more sophisticated microprocessors that already provide mechanisms that can be extended for multithreading, and with new multithreading techniques being developed to combine multithreading with instruction-level parallelism (by exploiting instruction-level parallelism both within and across threads, as discussed later in Section 11.8.5), this trend may change in the future.

Let us begin with the simple form of multithreading that we considered in message passing, in which instructions are executed from one thread until that thread encounters a long-latency event, at which point it is switched out and another thread switched in. The state of a thread —which includes the processor registers, the program counter (which holds the address of the next instruction to be executed in that process) the stack pointer, and some per-process parts of the processor status words (e.g. the condition codes)—is called the context of that thread, so multithreading is also called multiple-context processing. If the latency that we are trying to tolerate is large enough, then we can save the state to memory in software when the thread is switched out and then load it back into the appropriate registers when the thread is switched back in (note that restoring the processor status word requires operating system involvement). This is how multithreading is typically orchestrated on message-passing machines, so a standard single-threaded microprocessor can be used in that case. In a hardware-supported shared address space, and even more so on a uniprocessor, the latencies we are trying to tolerate are not that high. The overhead of saving and restoring state in software may be too high to be worthwhile, and we are likely to require hardware support. The cost of a context switch may also involve flushing or squashing instructions already in the processor pipeline, as we shall see. Let us examine this relationship between switch overhead and latency a little more quantitatively.

Consider processor utilization, i.e. the fraction of time that a processor spends executing useful instructions rather than stalled for some reason. The time threads spends executing before they encounter a long-latency event is called the *busy* time. The total amount of time spent switching among threads is called the *switching* time. If no other thread is ready, the processor is stalled until one becomes ready or the long-latency event it stalled on completes. The total amount of time spent stalled for any reason is called the *idle* time. The utilization of the processor can then be expressed as:

$$utilization = \frac{busy}{busy + switching + idle}.$$

(EQ 11.2)

It is clearly important to keep the switching cost low. Even if we are able to tolerate all the latency through multithreading, thus removing idle time completely, utilization and hence performance is limited by the time spent context switching. (Of course, processor utilization is used here just for illustrative purposes; as discussed in Chapter 4 it is not in itself a very good performance metric).

11.8.1 Techniques and Mechanisms

For processors that issue instructions from only a single thread in a given cycle, hardware-supported multithreading falls broadly into two categories, determined by the decision about *when to switch threads*. The approach assumed so far—in message passing, in multiprogramming to tolerate disk latency, and at the beginning of this section—is to let a thread run until it encounters a long-latency event (e.g. a cache miss, a synchronization event, or a high-latency instruction such as a divide), and then switch to another ready thread. This has been called the *blocked* approach, since a context switch happens only when a thread is blocked or stalled for some reason. Among shared address space systems, it is used in the MIT Alewife prototype [ABC+95]. The other major hardware-supported approach is to simply switch threads every processor cycle, effectively interleaving the processor resource among a pool of ready threads at a one-cycle granularity. When a thread encounters a long-latency event, it is marked as not being ready, and does not participate in the switching game until that event completes and the thread joins the ready pool again. Quite naturally, this is called the *interleaved* approach. Let us examine both approaches in some detail, looking both at their qualitative features and tradeoffs as well as their quantitative evaluation and implementation details. After covering blocked and interleaved multithreading for processors that issue instructions from only a single thread in a cycle, we will examine the integration of multithreading with instruction-level (superscalar) parallelism, which has the potential to overcome the limitations of the traditional approaches (see Section 11.8.5). Let us begin with the blocked approach.

Blocked Multithreading

The hardware support for blocked multithreading usually involves maintaining multiple hardware register files and program counters, for use by different threads. An *active thread*, or a context, is a thread that is currently assigned one of these hardware copies. The number of active threads may be smaller than the number of *ready threads* (threads that are not stalled but are ready to run), and is limited by the number of hardware copies of the resources. Let us first take a high-level look at the relationship among the latency to be tolerated, the thread switching overhead and the number of active threads, by analyzing an average-case scenario.

Suppose a processor can support *N* active threads at a time (*N*-way multithreading). And suppose that each thread operates by repeating the following cycle:

- execute useful instructions without stalling for *R* cycles (*R* is called the *run length*)
- encounter a high-latency event and switch to another thread.

Suppose that the latency we are trying to tolerate each time is *L* cycles, and the overhead of a thread switch is *C* cycles. Given a fixed set of values for *R*, *L* and *C*, a graph of processor utilization versus the number of threads *N* will look like that shown in Figure 11-25. We can see that
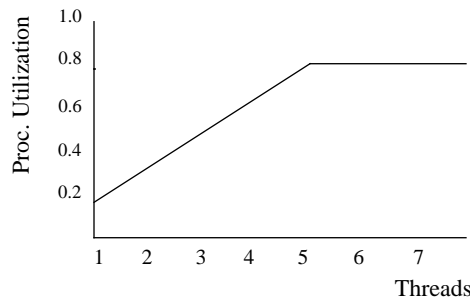

Figure 11-25 Processor Utilization versus number of threads in multithreading.

The figure shows the two regimes of operation: the linear regime, and saturation. It assumes R=40, L=200 and C=10 cycles.

there are two distinct regimes of operation. The utilization increases linearly with the number of threads up to a threshold, at which point it saturates. Let us see why.

Initially, increasing the number of threads allows more useful work to be done in the interval L that a thread is stalled, and latency continues to be hidden. Once *N* is sufficiently large, by the time we cycle through all the other threads—each with its run length of *R* cycles and switch cost of *C* cycles—and return to the original thread we may have tolerated all *L* cycles of latency. Beyond this, there is no benefit to having more threads since the latency is already hidden. The value of *N* for which this saturation occurs is given by *(N-1) R + NC = L*, or $N_{sat} = \dfrac{R + L}{R + C}$.

Beyond this point, the processor is always either busy executing a run or incurring switch overhead, so the utilization according to Equation 11.2 is:

$$u_{sat} = \frac{R}{R + C} = \frac{1}{1 + \dfrac{C}{R}} . \qquad \textbf{(EQ 11.3)}$$

If *N* is not large enough relative to *L*, then the runs of all *N-1* other threads will complete before the latency *L* passes. A processor therefore does useful work for *R+(N-1)*R* cycles out of every *R+L* and is idle for the rest of the time. That is, it is busy for *R+(N-1)*R* or *NR* cycles, switching for *N*C* cycles and idle for *L-[(N-1)R+N*C]* cycles, i.e. busy for *NR* out of every *R+L* cycles, leading to a utilization of

$$u_{lin} = \frac{NR}{R + L} = N \cdot \frac{1}{1 + \dfrac{L}{R}} . \qquad \textbf{(EQ 11.4)}$$

This analysis is clearly simplistic, since it uses an average run length of R cycles and ignores the burstiness of misses and other long-latency events. Average-case analysis may lead us to assume that less threads suffice than are necessary to handle the bursty situations where latency tolerance may be most crucial. However, the analysis suffices to make the key points. Since the best utilization we can get with any number of threads, u_{sat}, decreases with increasing switch cost C (see Equation 11.3), it is very important that we keep switch cost low. Switch cost also affects the types of latency that we can hide; for example, pipeline latencies or the latencies of misses that are satisfied in the second-level cache may be difficult to hide unless the switch cost is very low.

Switch cost can be kept low if we provide hardware support for several active threads; that is, have a hardware copy of the register file and other processor resources needed to hold thread state for enough threads to hide the latency. Given the appropriate hardware logic, then, we can simply switch from one set of hardware state to another upon a context switch. This is the approach taken in most hardware multithreading proposals. Typically, a large register file is either statically divided into as many equally sized register frames as the active threads supported (called a segmented register file), or the register file may be managed as a sort of dynamically managed cache that holds the registers of active contexts (see Section 11.8.3). While replicating context state in hardware can bring the cost of this aspect of switching among active threads down to a single cycle—it's like changing a pointer instead of copying a large data structure—there is another time cost for context switching that we have not discussed so far. This cost arises from the use of pipelining in instruction execution.

When a long-latency event occurs, we want to switch the current thread out. Suppose the long latency event is a cache miss. The cache access is made only in the data fetch stage of the instruction pipeline, which is quite late in the pipeline. Typically, the hit/miss result is only known in the writeback stage, which is at the end of the pipeline. This means that by the time we know that a cache miss has occurred and the thread should be switched out, several other instructions from that thread (potentially k, where k is the pipeline depth) have already been fetched and are in the pipeline (see Figure 11-26). We are faced with three possibilities:

| IF1 | IF2 | RF | EX | DF1 | DF2 | WB |

| A_{i+6} | A_{i+5} | A_{i+4} | A_{i+3} | A_{i+2} | A_{i+1} | A_i |
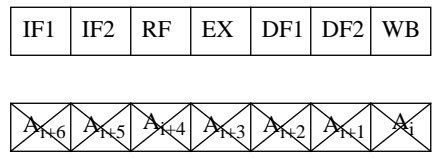
Figure 11-26 Impact of late miss detection in a pipeline.

Thread A is the current thread running on the processor. A cache miss occurring on instruction A_i from this thread is only detected after the second data fetch stage (DF2) of the pipeline, i.e. in the writeback (WB) cycle of A_i's traversal through the pipeline. At this point, the following six instructions from thread A (A_{i+1} through A_{i+6}) are already in the different stages of the assumed seven-stage pipeline. If all the instructions are squashed when the miss is detected (dashed x's), we lose at least seven cycles of work.

- Allow these subsequent instructions to complete, and start to fetch instructions from the new thread at the same time.
- Allow the instructions to complete before starting to fetch instructions from the new thread.
- Squash the instructions from the pipeline and then start fetching from the new thread.

The first case is complex to implement, for two reasons. First, instructions from different threads will be in the pipeline at the same time. This means that the standard uniprocessor pipeline registers and dependence resolution mechanisms (interlocks) do not suffice. Every instruction must be tagged with its context as it proceeds through the pipeline, and/or multiple sets of pipeline registers must be used to distinguish results from different contexts. In addition to the increase in area (the pipeline registers are a substantial component of the data path) and the fact that the processor has to be changed much more substantially, the use of multiple pipeline registers at the pipeline stages means that the registers must be multiplexed onto the latches for the next stage, which may increase the processor cycle time. Since part of the motivation for the blocked scheme is its design simplicity and ability to use commodity processors with as little design effort and modification as possible, this may not be a very appropriate choice. The second problem with this choice is that the instructions in the pipeline following the one that incurs the cache miss may stall the pipeline because they may depend on the data returned by the miss.

The second case avoids having instructions from multiple threads simultaneously in the pipeline. However, it does not do anything about the second problem above. The third case avoids both problems: Instructions from the new thread do not interact with those from the old one since the latter are squashed, and the instructions being squashed means they will not miss and hold up the pipeline. This third choice is the simplest to implement, since the standard uniprocessor pipeline suffices and already has the ability to squash instructions, and is the favored choice for the blocked scheme.

How does a context switch get triggered in the blocked approach to hide the different kinds of latency. On a cache miss, the switch can be triggered by the detection of the miss in hardware. For synchronization latency, we can simply ensure that an explicit context switch instruction follows every synchronization event that is expected to incur latency (or even all synchronization events). Since the synchronization event may be satisfied by another thread on the same processor, an explicit switch is necessary in this case to avoid deadlock. Long pipeline stalls can also be handled by inserting a switch instruction following a long-latency instruction such as a divide. Finally, short pipeline stalls like data hazards are likely to be very difficult to hide with the blocked approach.

To summarize, the blocked approach has the following positive qualities:

- relatively low implementation cost (as we shall see in more detail later)
- good single-thread performance: If only a single thread is used per processor, there are no context switches and this scheme performs just like a standard uniprocessor would.

The disadvantage is that the context switch overhead is high: approximately the depth of the pipeline, even when registers and other processor state do not have to be saved to or restored from memory. This overhead limits the types of latencies that can be hidden. Let us examine the performance impact through an example, taken from [LGH94].

Example 11-3   Suppose four threads, *A, B, C, and D,* run on a processor. The threads have the following activity:

A issues two instructions, with the second instruction incurring a cache miss, then issues four more.

B issues one instruction, followed by a two-cycle pipeline dependency, followed by two more instructions, the last of which incurs a cache miss, followed by two more.

C issues four instructions, with the fourth instruction incurring a cache miss, followed by three more.

D issues six instructions, with the sixth instruction causing a cache miss, followed by one more.

Show how successive pipeline slots are either occupied by threads or wasted in a blocked multithreaded execution. Assume a simple four-deep pipeline and hence four-cycle context-switch time, and a cache miss latency of ten cycles (small numbers are used here for ease of illustration).
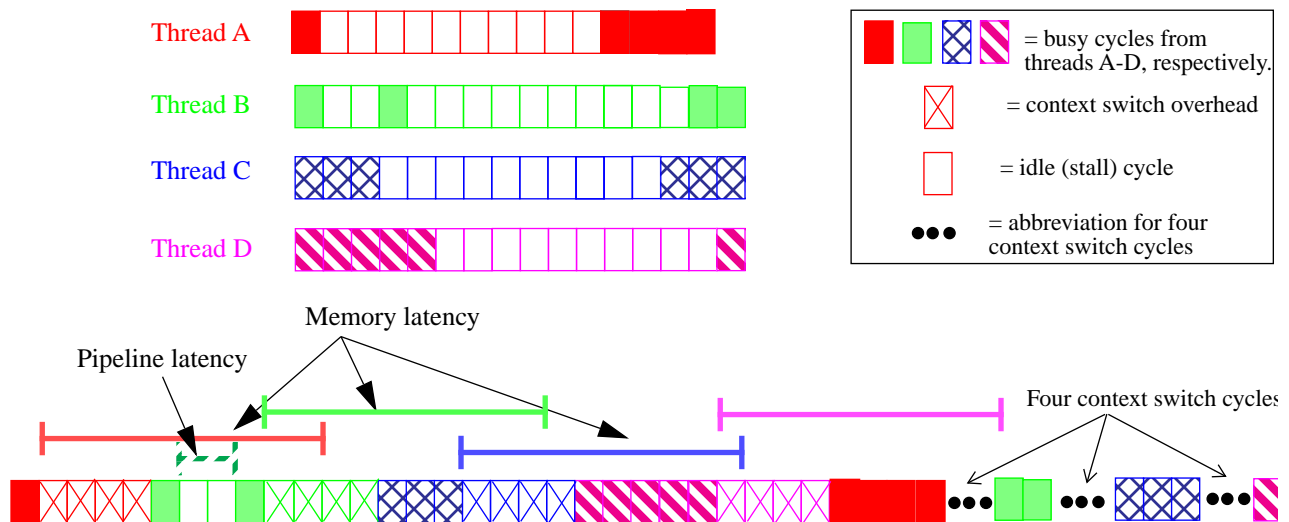

Figure 11-27 Latency tolerance in the blocked multithreading approach.

The top part of the figure shows how the four active threads on a processor would behave if each was the only thread running on the processor. For example, the first thread issues one instruction, then issues a second one which incurs a cache miss, then issues some more. The bottom part shows how the processor switches among threads when they incur a cache miss. In all cases, the instruction shown in each slot is the one that is (or would be) in the last, WB stage of the pipeline in that cycle. Thus, if the second instruction from the first thread had not missed, it would have been in the WB state in the second cycle shown at the bottom. However, since it misses, that cycle is wasted and the three other instructions that have already been fetched into the 4-deep pipeline (from the same thread) have to be squashed in order to switch contexts. Note that the two-cycle pipeline latency does not generate a thread switch, since the switch overhead of four cycles is larger than this latency.

Answer   The solution is shown in Figure 11-27, assuming that threads are chosen round-robin starting from thread *A*. We can see that while most of the memory latency is hidden, this is at the cost of context switch overhead. Assuming the pipeline is in steady state at the beginning of this sequence, we can count cycles starting from the time the first instruction reaches the WB stage (i.e. the first cycle shown for the multithreaded execution in the bottom part of the figure). Of the 51 cycles taken in the multithreaded execution, 21 are useful busy cycles, two are pipeline stalls, there are *no* idle cycles stalled on memory, and 28 are context switch cycles, leading to a

processor utilization of (21/51) * 100 or only 41% despite the extremely low cache miss penalty assumed.

Interleaved Multithreading

In the interleaved approach, in every processor clock cycle a new instruction is chosen from a different thread that is ready and active (assigned a hardware context) so that threads are switched every cycle rather than only on long-latency events. When a thread incurs a long-latency event, it is simply disabled or removed from the pool of ready threads until that event completes and the thread is labeled ready again. The key advantage of the interleaved scheme is that there is no context switch overhead. No event needs to be detected in order to trigger a context switch, since this is done every cycle anyway. Segmented or replicated register files are used here as well to avoid the need to save and restore registers. Thus, if there are enough concurrent threads, in the best case all latency will be hidden without any switch cost, and every cycle of the processor will perform useful work. An example of this ideal scenario is shown in Figure 11-28,
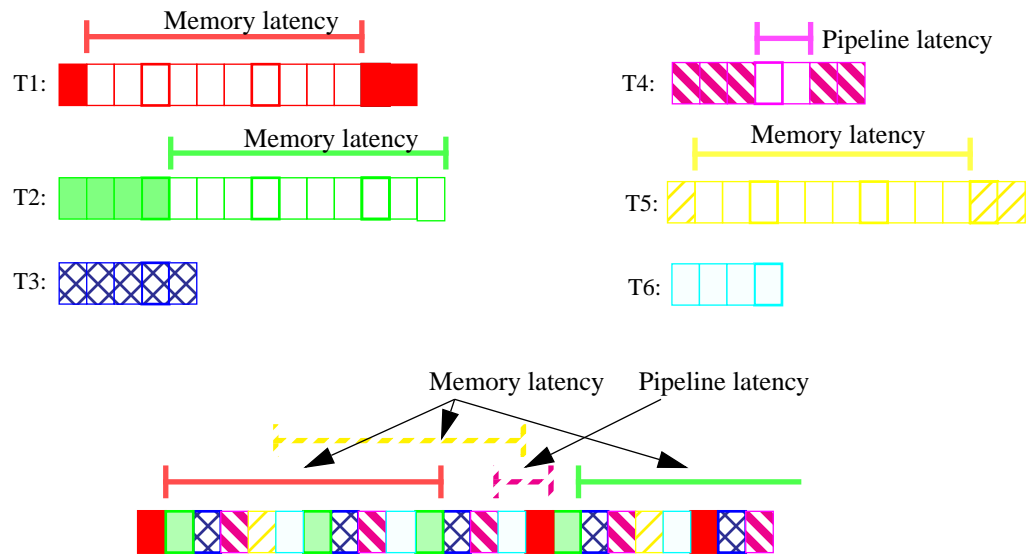
Figure 11-28 Latency tolerance in an ideal setting the interleaved multithreading approach.

The top part of the figure shows how the six active threads on a processor would behave if each was the only thread running on the processor. The bottom part shows how the processor switches among threads round-robin every cycle, leaving out those threads whose last instruction caused a stall that has not been satisfied yet. For example, the first (solid) thread is not chosen again until its high-latency memory reference is satisfied and its turn comes again in the round-robin scheme.

where we assume six active threads for illustration. The typical disadvantage of the interleaved approach is the higher hardware cost and complexity, though the specifics of this and other potential disadvantages depend on the particular type of interleaved approach used.

Interleaved schemes have undergone a fair amount of evolution. The early schemes severely restricted the number and type of instructions from a given thread that could be in the processor's pipeline at a time. This reduced the need for hardware interlocks and simplifying processor

design, but had severe implications for the performance of single-threaded programs. More recent interleaved schemes greatly reduce these restrictions (blocked schemes of course do not have them, since the same thread runs until it encounters a long-latency event). In practice, another distinguishing feature among interleaved schemes is whether they use caches to reduce latency before trying to hide it. Machines built so far using the interleaved technique do not use caches at all, but rely completely on latency tolerance through multithreading [Smi81, ACC+90]. More recent proposals advocate the use of interleaved techniques and full pipeline interlocks with caches. [LGH94] (recall that blocked multithreaded systems use caching since they want to keep a thread running efficiently for as long as possible). Let us look at some interesting points in this development.

The Basic Interleaved Scheme

The first interleaved multithreading scheme was that used in the Denelcor HEP (Heterogeneous Element Processor) multiprocessor, developed between 1978 and 1985 [Smi81]. Each processor had up to 128 active contexts, 64 user-level and 64 privileged, though only about 50 of them were available to the user. The large number of active contexts was needed even though the memory latency was small—about 20-40 cycles without contention—since the machine had no caches so the latency was incurred on every memory reference (memory modules were all on the other side of a multistage interconnection network, but the processor had an additional direct connection to one of these modules which it could consider its "local" module). The 128 active contexts were supported by replicating the register file and other critical state 128 times. The pipeline on the HEP was 8 cycles deep. The pipeline supported interlocks among non-memory instructions, but did not allow more than one memory, branch or divide operation to be in the pipeline at a given time. This meant that several threads had to be active on each processor at a time just to utilize the pipeline effectively, even without any memory or other stalls. For parallel programs, for which HEP was designed, this meant that the degree of explicit concurrency in the program had to be much larger than the number of processors, especially in the absence of caches to reduce latency, greatly restricting the range of applications that would perform well.

Better Use of the Pipeline

Recognizing the drawbacks or poor single-thread performance and very large number of needed threads in allowing only a single memory operation from a thread in the pipeline at a time, systems descended from the HEP have alleviated this restriction. These systems include the Horizon [KuS88] and the Tera [ACC+90] multiprocessors. They still do not use caches to reduce latency, relying completely on latency tolerance for all memory references.

The first of these designs, the Horizon, was never actually built. The design allows multiple memory operations from a thread to be in the pipeline, but does this without hardware pipeline interlocks even for non-memory instructions. Rather, the analysis of dependences is left to the compiler. The idea is quite simple. Based on the compiler analysis every instruction is tagged with a three-bit "lookahead" field, which specifies the number of immediately following instructions in that thread that are independent of that instruction. Suppose the lookahead field for an instruction is five. This means that the next five instructions (memory or otherwise) are independent of the current instruction, and so can be in the pipeline with the current instruction even though there are no hardware interlocks to resolve dependences. Thus, when a long latency event is encountered, the thread does not immediately become unready but can issue five more instructions before it becomes unready. The maximum value of the lookahead field is seven, so if more

than seven following instructions are independent, the machine cannot know this and will prohibit more than seven from entering the pipeline until the current one leaves.

Each "instruction" or cycle in Horizon can in fact issue up to three operations, one of which may be a memory operation. This means that twenty-one independent operations must be found to achieve the maximum lookahead, so sophisticated instruction scheduling by the compiler is clearly very important in this machine. It also means that for a typical program it is very likely that a memory operation is issued every instruction or two. Since the maximum lookahead size is larger than the average distance between memory operations, allowing multiple memory operations in the pipe is very important to hiding latency. However, in the absence of caches every memory operation is a long-latency event, now occurring almost once per instruction, so a large number of ready threads is still needed to hide latency. In particular, single-thread performance—the performance of programs that are not multithreaded—is not helped much by a small amount of lookahead without caches. The small number of lookahead bits provided is influenced by the high premium on bits in the instruction word and particularly by the register pressure introduced by having three results per instruction times the number of lookahead instructions; more lookahead and greater register pressure might have been counter-productive for instruction scheduling. Effective use of large lookahead also requires more sophistication in the compiler.

The Tera architecture is the latest in the series of interleaved mutithreaded architectures without caches. It is actually being built by Tera Computer Company. Tera manages instruction dependences differently than Horizon and HEP: It provides hardware interlocks for instructions that don't access memory (like HEP), and Horizon-like lookahead for memory instructions. Let us see how this works.

The Tera machine separates operations into memory operations, "arithmetic" operations, and control operations (e.g. branches). The unusual, custom-designed processor can issue three operations per instruction, much like Horizon, either one from each category or two arithmetic and one memory operation. Arithmetic and control operations go into one pipeline, which unlike Horizon has hardware interlocks to allow multiple operations from the same thread. The pipeline is very deep, and there is a sizeable minimum issue delay between consecutive instructions from a thread (about 16 cycles, with the pipeline being deeper than this), even if there are no dependences between consecutive instructions. Thus, while more than one instruction can be in this pipeline at the same time, several interleaved threads (say 16) are required to hide the instruction latency. Even without memory references, a single thread would at best complete one instruction every 16 cycles.

Memory operations pose a bigger problem, since they incur substantially larger latency. Although Tera uses very aggressive memory and network technology, the average latency of a memory reference without contention is about 70 cycles (a processor cycle is 2.8ns). Tera therefore uses compiler-generated lookahead fields for memory operations. Every instruction that includes a memory operation (called a memory instruction) has a 3-bit lookahead field which tells how many immediately following instructions (memory or otherwise) are independent of that memory operation. Those instructions do not have to be independent of one another (those dependences are taken care of by the hardware interlocks in that pipeline), just of that memory operation. The thread can then issue that many instructions past the memory instruction before it has to render itself unready. This change in the meaning of lookahead makes it easier for the compiler to schedule instructions to have larger lookahead values, and also eases register pressure. To make this concrete, let us look at an example.

Example 11-4   Suppose that the minimum issue delay between consecutive instructions from a thread in Tera is 16, and the average memory latency to hide is 70 cycles. What is the smallest number of threads that would be needed to hide latency completely in the best case, and how much lookahead would we need per memory instruction in this case?

Answer   A minimum issue delay of 16 means that we need about 16 threads to keep the pipeline full. If 16 threads were to suffice to hide 70 cycles of memory latency, and since memory operations are almost one per instruction in each thread, we would like each of the sixteen threads to issue about 4 independent instructions before it is made unready after a memory operation, i.e. to have a lookahead of about 4. Since latencies are in fact often larger than the average uncontended 70 cycles, a higher lookahead of at most seven (three bits) is provided. Longer latencies would ask for larger lookahead values and sophisticated compilers. So would a desire for fewer threads, though this would require reducing the minimum issue delay in the non-memory pipeline as well.

While it seems from the above that supporting a few tens of active threads should be enough, Tera like HEP supports 128 active threads in hardware. For this, it replicates all processor state (program counter, status word and registers) 128 times, resulting in a total of 4096 64-bit general registers (32 per thread) and 1024 branch target registers (8 per thread). The large number of threads is supported for several reasons, and reflects the fundamental reliance of the machine on latency tolerance rather than reduction. First, some instructions may not have much lookahead at all, particularly read instructions and particularly when there are three operations per instruction. Second, with most memory references going into the network they may incur contention, in which case the latency to be tolerated may be much longer than 70 cycles. Third, the goal is not only to hide instruction and memory latency but also tolerate synchronization wait time, which is caused by load imbalance or contention for critical resources, and which is usually much larger than remote memory references.

The designers of the Tera system and its predecessors take a much more radical view to the redesign of the multiprocessor building block than advocated by the other latency tolerating techniques we have seen so far. The belief here is that the commodity microprocessor building block, with its reliance on caches and support for only one or a small number of hardware contexts, is inappropriate for general-purpose multiprocessing. Because of the high latencies and physically distributed memory in modern convergence architectures, the use of relatively "latency-intolerant" commodity microprocessors as the building block implies that much attention must be paid to data locality in both the caches and in data distribution. This makes the task of programming for performance too complicated, since compilers have not yet succeeded in managing locality automatically in any but the simplest cases and their potential is unclear. The Tera approach argues that the only way to make multiprocessing truly general purpose is to take this burden of locality management off software and place it on the architecture in the form of much greater latency tolerance support in the processor. Unlike the approach we have followed so far—reduce latency first, then hide the rest—this approach does not pay much attention to reducing latency at all; the processor is redesigned with a primary focus on tolerating the latency of fine-grained accesses through interleaved multithreading. If this technique is successful, the programmer's view of the machine can indeed be a PRAM (i.e. memory references cost a single cycle regardless of where they are satisfied, see Chapter 3) and the programmer can concentrate on concurrency rather than latency management. Of course, this approach sacrifices the tremendous

leverage obtained by using commodity microprocessors and caches, and faces head-on the challenge of the enormous effort that must be invested in the design of a modern high-performance processor. It is also likely to result in poor single-thread performance, which means that even uniprocessor applications must be heavily multithreaded to achieve good performance.

Full Single-thread Pipeline Support and Caching

While the HEP and the blocked multithreading approaches are in some ways at opposite ends of a multithreading spectrum, both have several limitations. The Tera approach improves matters from the basic HEP approach, but it still does not provide full single-thread support and requires many concurrent threads for good utilization. Since it does not use caches, every memory operation is a long-latency operation, which increases the number of threads needed and the difficulty of hiding the latency. This also means that every memory reference consumes memory and perhaps communication bandwidth, so the machine must provide tremendous bandwidth as well.

The blocked multithreading approach, on the other hand, requires less modification to a commodity microprocessor. It utilizes caches and does not switch threads on cache hits, thus providing good single-thread performance and needing a smaller number of threads. However, it has high context switch overhead and cannot hide short latencies. The high switch overhead also makes it less suited to tolerating the not-so-large latencies on uniprocessors than the latencies on multiprocessors. It is therefore difficult to justify either of these schemes for uniprocessors and hence for the high-volume marketplace.

It is possible to use an interleaved approach with both caching and full single-thread pipeline support, thus requiring a smaller number of threads to hide memory latency, providing better support for uniprocessors, and incurring lower overheads. One such scheme has been studied in detail, and has in fact been termed the "interleaved" scheme [LGH94]. We shall describe it next.

From the HEP and Tera approaches, this interleaved approach takes the idea of maintaining a set of active threads, each with its own set of registers and status words, and having the processor select an instruction from one of the ready threads every cycle. The selection is usually simple, such as round-robin among the ready threads. A thread that incurs a long latency event makes itself unready until the event completes, as before. The difference is that the pipeline is a standard microprocessor pipeline, and has full bypassing and forwarding support so that instructions from the same thread can be issued in consecutive cycles; there is no minimum issue delay as in Tera. In the best case, with no pipeline bubbles due to dependences, a k-deep pipeline may contain k instructions from the same thread. In addition, the use of caches to reduce latency implies that most memory operations are not long latency events; a given thread is ready a much larger fraction of the time, so the number of threads needed to hide latency is kept small. For example, if each thread incurs a cache miss every 30 cycles, and a miss takes 120 cycles to complete, then only five threads (the one that misses and four others) are needed to achieve full processor utilization.

The overhead in this interleaved scheme arises from the same source as in the blocked scheme, which is the other scheme that uses caches. A cache miss is detected late in the pipeline, and if there are only a few threads then the thread that incurs the miss may have fetched other instructions into the pipeline. Unlike in the Tera, where the compiler guarantees through lookahead that any instruction entering the pipeline after a memory instruction is independent of the memory instruction, we must do something about these instructions. For the same reason as in the blocked scheme, the proposed approach chooses to squash these instructions, i.e. to mark them as not

being allowed to modify any processor state. Note that due to the cycle-by-cycle interleaving there are instructions from other threads interleaved in the pipeline, so unlike the blocked scheme not all instructions need to be squashed, only those from the thread that incurred the miss. The cost of making a thread unready is therefore typically much smaller than that of a context switch in the blocked scheme, in which case all instructions in the pipeline must be squashed. In fact, if enough ready threads are supported and available, which requires a larger degree of state replication that is advocated by this approach, then there may not be multiple instructions in the pipeline from the missing thread and no instructions will need to be squashed. The comparison with the blocked approach is shown in Figure 11-29.
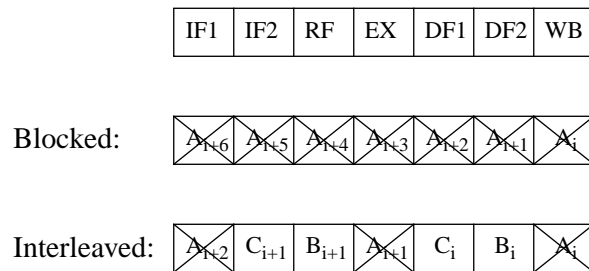
IF1	IF2	RF	EX	DF1	DF2	WB

Blocked:

A_{i+6}	A_{i+5}	A_{i+4}	A_{i+3}	A_{i+2}	A_{i+1}	A_i

Interleaved:

A_{i+2}	C_{i+1}	B_{i+1}	A_{i+1}	C_i	B_i	A_i

Figure 11-29 "Switch cost" in the interleaved scheme with full single-thread support.

The figure shows the assumed seven-stage pipeline, followed by the impact of late miss detection in the blocked scheme (taken from Figure 11-26) in which all instructions in the pipeline are from thread A and have to be squashed, followed by the situation for the interleaved scheme. In the interleaved scheme, instructions from three different threads are in the pipeline, and only those from thread A need to be squashed. The "switch cost" or overhead incurred on a miss is three cycles rather than seven in this case.

The result of the lower cost of making a thread unready is that shorter latencies such as local memory access latencies can be tolerated more easily than in the blocked case, making this interleaved scheme more appropriate for uniprocessors. Very short latencies that cannot be hidden by the blocked scheme, such as those of short pipeline hazards, are usually hidden naturally by the interleaving of contexts without even needing to make a context unready. The effect of the differences on the simple four-thread, four-deep pipeline example that we have been using so far is illustrated in Figure 11-30. In this simple example, assuming the pipeline is in steady state at the beginning of this sequence, the processor utilization is 21 cycles out of the 30 cycles taken in all, or 70% (compared to 41% for the blocked scheme example). While this example is contrived and uses unrealistic parameters for ease of graphical illustration, the fact remains that on modern superscalar processors that issue a memory operation in almost every cycle, the context switch overhead of switching on every cache miss may become quite expensive. The disadvantage of this scheme compared to the blocked scheme is greater implementation complexity.

The blocked scheme and this last interleaved scheme (henceforth called *the interleaved scheme*) start with simple, commodity processors with full pipeline interlocks and caches, and modify them to make them multithreaded. As stated earlier, even if they are used with superscalar processors, they only issue instructions from within a single thread in a given cycle. A more sophisticated multithreading approach exists for superscalar processors, but for simplicity let us examine the performance and implementation issues for these simpler, more directly comparable approaches first.
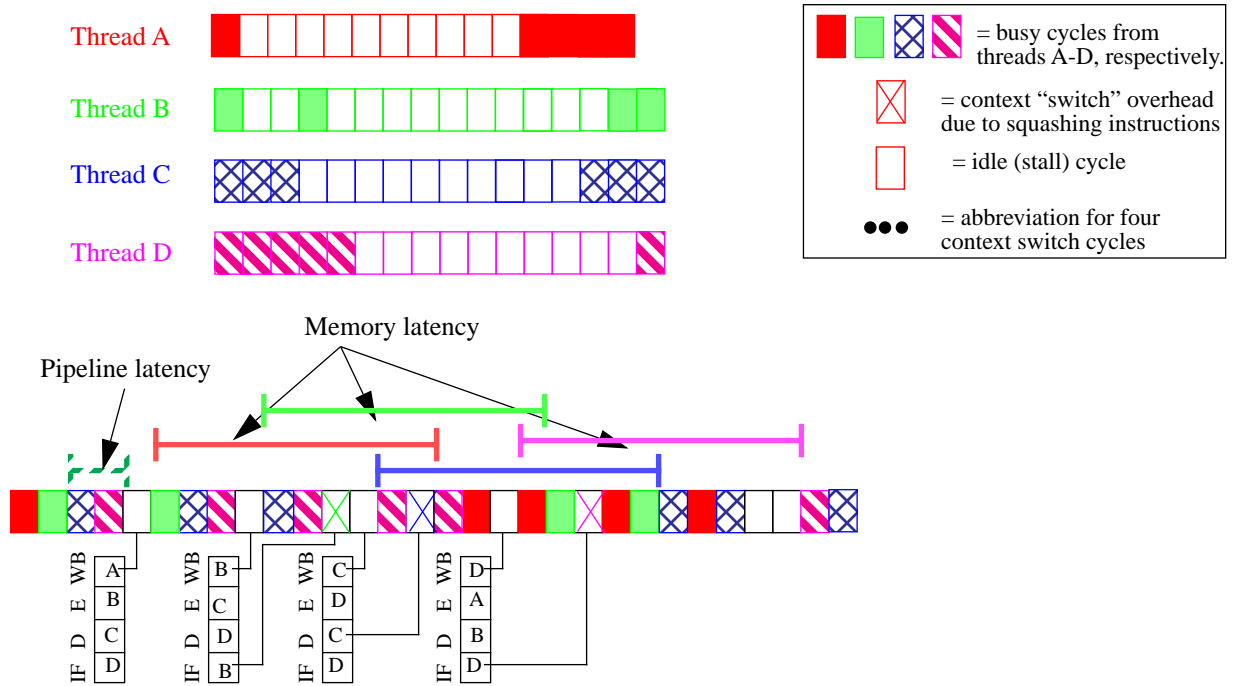
Figure 11-30 Latency tolerance in the interleaved scheme.

The top part of the figure shows how the four active threads on a processor would behave if each was the only thread running on the processor. The bottom part shows how the processor switches among threads. Again, the instruction in a slot (cycle) is the one that retires (or would retire) from the pipeline that cycle. In the first four cycles shown, an instruction from each thread retires. In the fifth cycle, A would have retired but discovers that it has missed and needs to become unready. The three other instructions in the pipeline at that time are from the three other threads, so there is no switch cost except for the one cycle due to instruction that missed. When B's next instruction reaches the WB stage (cycle 9 in the figure) it detects a miss and has to become unready. At this time, since A is already unready an instruction each from C and D have entered the pipeline as has one more from B (this one is now in its IF stage, and would have retired in cycle 12). Thus, the instruction from B that misses wastes a cycle, and one instruction from B has to be squashed. Similarly, C's instruction that would have retired in cycle 13 misses and causes another instruction from C to be squashed, and so on.

11.8.2 Performance Benefits

Studies have shown that both the blocked scheme and the interleaved scheme with full pipeline interlocks and caching can hide read and write latency quite effectively [LGH94, KCA91]. The number of active contexts needed is found to be quite small, usually in the vicinity of four to eight, although this may change as the latencies become longer relative to processor cycle time.

Let us examine some results from the literature for parallel programs [Lau94]. The architectural model is again a cache-coherent multiprocessor with 16 processors using a memory-based protocol. The processor model is single issue and modeled after the MIPS R4000 for the integer pipeline and the DEC Alpha 21064 for the floating-point pipeline. The cache hierarchy used is a small (64 KB) single-level cache, and the latencies for different types of accesses are modeled after the Stanford DASH prototype [LLJ+92]. Overall, both the blocked and interleaved schemes were found to improve performance substantially. Of the seven applications studied, the speedup from multithreading ranged from 2.0 to nearly 3.5 for three applications, from 1.2 to 1.6 for three oth-

ers, and was negligible for the last application because it had very little extra parallelism to begin with (see Figure 11-31). The interleaved scheme was found to outperform the blocked scheme, as
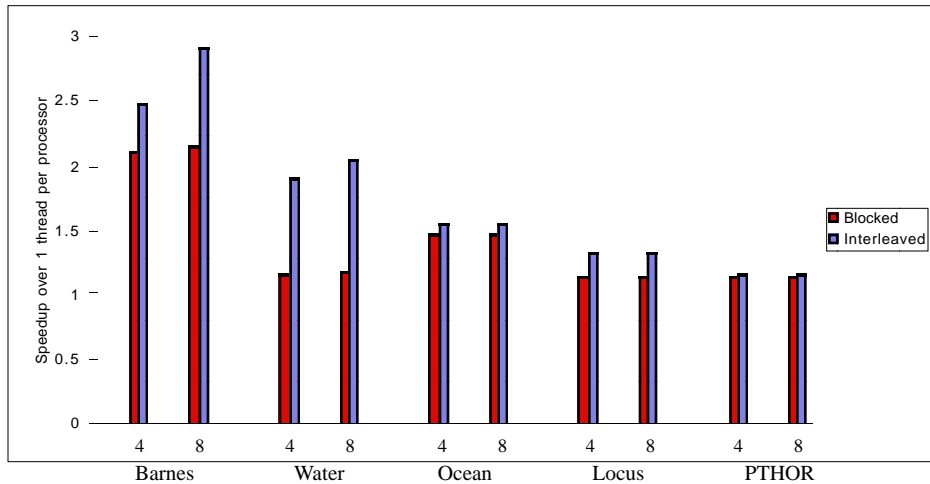

Figure 11-31 Speedups for blocked and interleaved multithreading.

The bars show speedups for different numbers of contexts (4 and 8) relative to a single-context per processor execution for seven applications. All results are for sixteen processor executions. Some of the applications were introduced in Figure 11-24 on page 810. Water is a molecular dynamics simulation of water molecules in liquid state. The simulation model assumes a single-level, 64KB, direct-mapped writeback cache per processor. The memory system latencies assumed are 1 cycle for a hit in the cache, 24-45 cycles with a uniform distribution for a miss satisfied in local memory, 75-135 cycles for a miss satisfied at a remote home, and 96-156 cycles for a miss satisfied in a dirty node that is not the home. These latencies are clearly low by modern standards. The R4000-like integer unit has a seven-cycle pipeline, and the floating point unit has a 9-stage pipeline (5 execute stages). The divide instruction has a 61 cycle latency, and unlike other functional units the divide unit is not pipelined. Both schemes switch contexts on a divide instruction. The blocked scheme uses an explicit switch instruction on synchronization events and divides, which has a cost of 3 cycles (less than a full 7-cycle context switch, because the decision to switch is known after the decode stage rather than at the writeback stage). The interleaved scheme uses a backoff instruction in these cases, which has a cost of 1-3 cycles depending on how many instructions need to be squashed as a result.

expected from the discussion above, with a geometric mean speedup of 2.75 compared to 1.9.

The advantages of the interleaved scheme are found to be greatest for applications that incur a lot of latency due to short pipeline stalls, such as those for result dependences in floating point add, subtract and multiply instructions. Longer pipeline latencies, such as the tens of cycle latencies of divide operations, can be tolerated quite well by both schemes, though the interleaved scheme still performs better due to its lower switch cost. The advantages of the interleaved scheme are retained even when the organizational and performance parameters of the extended memory hierarchy were changed (for example, longer latencies and multilevel caches). They are in fact likely to be even greater with modern processors that issue multiple operations per cycle, since the frequency of cache misses and hence context switches is likely to increase. A potential disadvantage of multithreading is that multiple threads of execution share the same cache, TLB and branch prediction unit, raising the possibility of negative intereference between them (e.g. mapping conflicts across threads in a low-associativity cache). These negative effects have been found to be quite small in published studies.

Figure 11-32 shows more detailed breakdowns of execution time, averaged over all processors, for two applications that illustrate interesting effects. With a single context, Barnes shows signif-
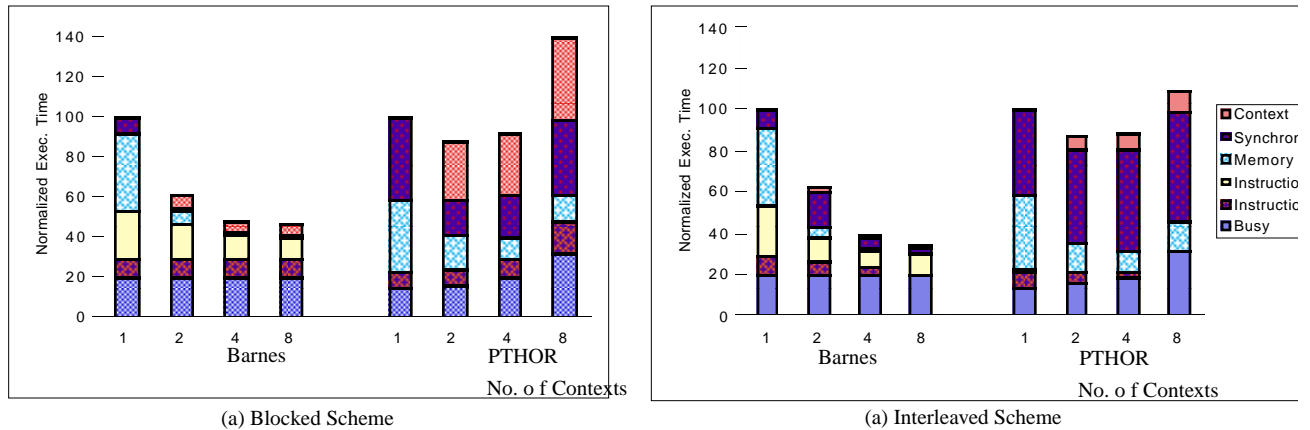
| (a) Blocked Scheme | (a) Interleaved Scheme |

Figure 11-32 Execution time breakdowns for two applications under multithreading.

Busy time is the time spent executing application instructions; pipeline stall time is the time spent stalled due to pipeline dependences; memory time and synchronization time are the time spent stalled on the memory system and at synchronization events, respectively. Finally, context switch time is the time spent in context switch overhead.

icant memory stall time due to the single-level direct-mapped cache used (as well as the small problem size of only 4K bodies). The use of more contexts per processor is able to hide most of the memory latency, and the lower switch cost of the interleaved scheme is clear from the figure. The other major form of latency in Barnes (and in Water) is pipeline stalls due to long-latency floating point instructions, particularly divides. The interleaved scheme is able to hide this latency more effectively than the blocked scheme. However, both start to taper off in their ability to hide divide latency at more than four contexts. This is because the divide unit is not pipelined, so it quickly becomes a resource bottlenecks when divides from different contexts compete. PTHOR is an example of an application in which the use of more contexts does not help very much, and even hurts as more contexts are used. Memory latency is hidden quite well as soon as we go to two contexts, but the major bottleneck is synchronization latency. The application simply does not have enough extra parallelism (slackness) to exploit multiple contexts effectively. Note that busy time increases with the number of contexts in PTHOR. This is because the application uses a set of distributed task queues, and more time is spent maintaining these queues as the number of threads increases. These extra instructions cause more cache misses as well.

Prefetching and multithreading can hide similar types of memory access latency, and one may be better than the other in certain circumstances. The performance benefits of using the two together are not well understood. For example, the use of multithreading may cause constructive or destructive interference in the cache among threads, which is very difficult to predict and so makes the analysis of what to prefetch more difficult.

The next two subsections discuss some detailed implementation issues for the blocked and interleaved schemes, examining the additional implementation complexity needed to implement each scheme beyond that needed for a commodity microprocessor. Readers not interested in implementation can skip to Section 11.8.5 to see a more sophisticated multithreading scheme for superscalar processors. Let us begin with the blocked scheme, which is simpler to implement.

11.8.3 Implementation Issues for the Blocked Scheme

Both the blocked and interleaved schemes have two kinds of requirements: *state replication*, and *control*. State replication, essentially involves replicating the register file, program counter and relevant portions of the processor status word once per active context, as discussed earlier. For control, logic and registers are needed to manage switching between contexts, making contexts ready and unready, etc. The PC unit is the portion of the processor that requires significant changes, so we discuss it separately as well.

State Replication

Let us look at the register file, the processor status word, and the program counter unit separately. Simple ways to give every active context its own set of registers are to replicate the register file or to use a larger, statically segmented register file (see Figure 11-33). While this allows registers to
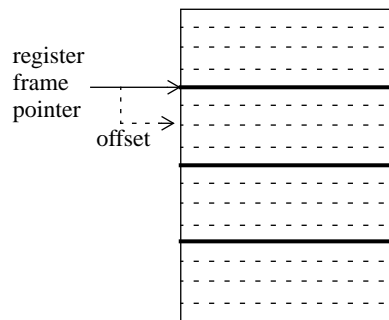

Figure 11-33 A segmented register file for a multithreaded processor.

The file is divided into four frames, assuming that four contexts can be active at a given time. The register values for each active context remain in that context's register frame across context switches, and each context's frame is managed by the compiler as if it were itself a register file. A register in a frame is accessed through the current frame pointer and a register offset within the frame, so the compiler need not be aware of which particular frame a context is in (which is determined at run time). Switching to a different active context only requires that the current frame pointer be changed.

be accessed quickly, it may not use silicon area efficiently. For example, since only one context runs at a time until it encounters a long-latency event, only one register file is actively being used for some time while the others are idle. At the very least, we would like to share the read and write ports across register files, since these ports often take up a substantial portion of the silicon area of the files. In addition, some contexts might require more or less registers than others, and the relative needs may change dynamically. Thus, allowing the contexts to share a large register file dynamically according to need may provide better register utilization than splitting the registers statically into equally-sized register files. This results in a cache-like structure, indexed by context identifier and register offset, with the potential disadvantage that the register file is larger so has a higher access time. Several proposals have been made to improve register file efficiency in one or more of the above issues [NuD95,Lau94,Omo94,Smi85], but substantial replication is needed in all cases. The MIT Alewife machine uses a modified Sun SPARC processor, and uses its register windows mechanism to provide a replicated register file.

A modern "processor status word" is actually several registers; only some parts of it (such as floating point status/control, etc.) contain process-specific state rather than global machine state, and only these parts need to be replicated. In addition, multithreading introduces a new global status word called the context status word (CSW). This contains an identifier that specifies which context is currently running, a bit that says whether context switching is enabled (we have seen that it may be disabled while exceptions are handled), and a bit vector that tells us which of the currently active contexts (the ones whose state is maintained in hardware) is ready to execute. Finally, TLB control registers need to be modified to support different address space identifiers from the different contexts, and to allow a single TLB entry to be used for a page that is shared among contexts.

PC Unit

Every active context must also have its program counter (PC) available in hardware. Processors that support exceptions efficiently provide a mechanism to do this with minimal replication, since in many ways exceptions behave like context switches. Here's how this works. In addition to the PC chain, which holds the PCs for the instructions that are in the different stages of the pipeline, a register called the exception PC (EPC) is provided in such processors. The exception EPC is fed by the PC chain, and contains the address of the last instruction retired from the pipeline. When an exception occurs, the loading of the EPC is stopped with the faulting instruction, all the incomplete instructions in the pipeline are squashed, and the exception handler address is put in the PC. When the exception returns, the EPC is loaded into the PC so the faulting instruction is reexecuted. This is exactly the functionality we need for multiple contexts. We simply need to replicate the EPC register, providing one per active context. The EPC for a context serves to handle both exceptions as well as context switches. When a given context is operating, the PC chain feeds into its EPC while the EPCs of other contexts simply retain their values. On an exception, the current context's EPC behaves exactly as it did in the single-threaded case. On a context switch, the current context's EPC stops being loaded at the faulting (long-latency) instruction and the incomplete instructions in the pipeline are squashed (as for an exception). The PC is loaded from the EPC of the selected next context, which then starts executing, and so on. The only drawback of this scheme is that now an exception cannot take a context switch (since the value loaded into the EPC by the context that incurred the exception will be lost), so context switches must be disabled when an exception occurs and re-enabled when the exception returns. However, the PCs can still be managed through software saving and restoring even in this case.

Control

The key functions of the control logic in a blocked implementation are to detect when to switch contexts, to choose the context to switch to, and to orchestrate and perform the switch. Let us discuss each briefly.

A context switch in the blocked scheme may be triggered by three events: (i) a cache miss, (ii) an explicit context switch instruction, used for synchronization events and very long-latency instructions, and (iii) a timeout. The timeout is used to ensure that a single context does not run too long or spin waiting on a flag that will therefore never be written. The decision to switch on a cache miss can be based on three signals: the cache miss notification, of course; a bit that says that context switching is enabled, and a signal that states that there is a context ready to run. A simple way to implement an explicit context switch instruction is to have it behave as if the next instruction generated a cache miss (i.e. to raise the cache miss signal or generate another signal that has the same effect on the context switch logic); this will cause the context to switch, and to be

restarted from that following instruction. Finally, the timeout signal can be generated via a resettable threshold counter.

Of the many policies that can be used to select the next context upon a switch, in practice simply switching to the next active context in a round-robin fashion—without concern for special relationships among contexts or the history of the contexts' executions—seems to work quite well. The signals this requires are the current context identifier, the vector of ready contexts, and the signal that detected the need to switch.

Finally, orchestrating a context switch in the blocked scheme requires the following actions. They are required to complete by different stages in the processor pipeline, so the control logic must enable the corresponding signals in the appropriate time windows.

- Save the address of the first uncompleted instruction from the current thread.
- Squash all incomplete instructions in the pipeline.
- Start executing from the (saved) PC of the selected context.
- Load the appropriate address space identifier in the TLB bound registers.
- Load various control/status registers from the (saved) processor status words of the new context, including the floating point control/status register and the context identifier.
- Switch the register file control to the register file for the new context, if applicable.

In summary, the major costs of implementing blocked context switching come from replicating and managing the register file, which increases both area and register file access time. If the latter is in the critical path of the processor cycle time, it may cause the pipeline depth to be increased to maintain a high clock rate, which can increase the time lost on branch mispredictions. The replication needed in the PC unit is small and easy to manage, focused mainly on replicating a single register for the exception PC. The number of process- or context-specific registers in the processor status words is also small, as is the number of signals needed to manage context switch control. Let us examine the interleaved scheme.

11.8.4 Implementation Issues for the Interleaved Scheme

A key reason that the blocked scheme is relatively easy to implement is that most of the time the processor behaves like a single-threaded processor, and invokes additional complexity and processor state changes only at context switches. The interleaved scheme needs a little more support, since it switches among threads every cycle. This potentially requires the processor state to be changed every cycle, and implies that the instruction issue unit must be capable of issuing from multiple active streams. A mechanism is also needed to make contexts active and inactive, and feed the active/inactive status into the instruction unit every cycle. Let us again look at the state replication and control needs separately.

State Replication

The register file must be replicated or managed dynamically, but the pressure on fast access to different parts of the entire register file is greater due to the fact that successive cycles may access the registers of different contexts. We cannot rely on the more gradually changing access patterns of the blocked scheme (the Tera processor uses a banked or interleaved register file, and a thread may be rendered unready due to a busy register bank as well). The parts of the process status

word that must be replicated are similar to those in the blocked scheme, though again the processor must be able to switch status words every cycle.

PC Unit

The greatest difference in changes is to the PC unit. Instructions from different threads are in the pipeline at the same time, and the processor must be able to issue instructions from different threads every cycle. Contexts become active and inactive dynamically, and this status must feed into the PC unit to prevent instructions being selected from an inactive context. The processor pipeline is also impacted, since to implement bypassing and forwarding correctly the processor's PC chain must now carry a context identifier for each pipeline stage. In the PC unit itself, new mechanisms are needed for handling context availability and for squashing instructions, for keeping track of the next instruction to issue, for handling branches, and for handling exceptions. Let us examine some of these issues briefly. A fuller treatment can be found in the literature [Lau94].

Consider context availability. Contexts become unavailable due to either cache misses or explicit backoff instructions that make the context unavailable for a specified number of cycles. The backoff instructions are issued for example at synchronization events. The issuing of further instructions from that context is stopped by clearing a "context available" signal. To squash the instructions already in the pipeline from that context, we must broadcast a squash signal as well as the context identifier to all stages, since we don't know which stages contain instructions from that context. On a cache miss, the address of the instruction that caused the miss is loaded into the EPC. Once the cache miss is satisfied and the context becomes available again, the PC bus is loaded from the EPC when that context is selected next. Explicit backoff instructions are handled similarly, except that we do not want the context to resume from the backoff instruction itself but rather from the instruction that follows it. To orchestrate this, a *next* bit can be included in the EPC.

There are three sources that can determine the next instruction to be executed from a given thread: the next sequential instruction, the predicted branch from the branch target buffer (BTB), and the computed branch if the prediction is detected to be wrong. When only a single context is in the pipeline at a time, the appropriate next instruction address can be driven onto the PC bus from the "next PC" (NPC) register as soon as it is determined. In the interleaved case, however, in the cycle when the next instruction address for a given context is determined and ready to be put on the PC bus, it may not be the context scheduled for that cycle. Further, since the NPC for a context may be determined in different pipeline stages for different instructions—for example, it is determined much later for a mispredicted branch than for a correctly predicted branch or a non-branch instruction—different contexts could produce their NPC value during the same cycle. Thus, the NPC value for each context must be held in a holding register until it is time to execute the next instruction from that context, at which point it will be driven onto the PC bus (see Figure 11-34).

Branches too require some additional mechanisms. First, the context identifier must be broadcast to the pipeline stages when squashing instructions past a mispredicted branch, but this is the same functionality needed when making contexts unavailable. Second, by the time the actual branch target is computed, the predicted instruction that was fetched speculatively could be anywhere in the pipeline or may not even have been issued yet (since other contexts will have intervened unpredictably). To find this instruction address to determine the correctness of the branch, it may be necessary for branch instructions to carry along with them their predicted address as they proceed along the pipeline stages. One way to do this is to provide a *predicted PC* register

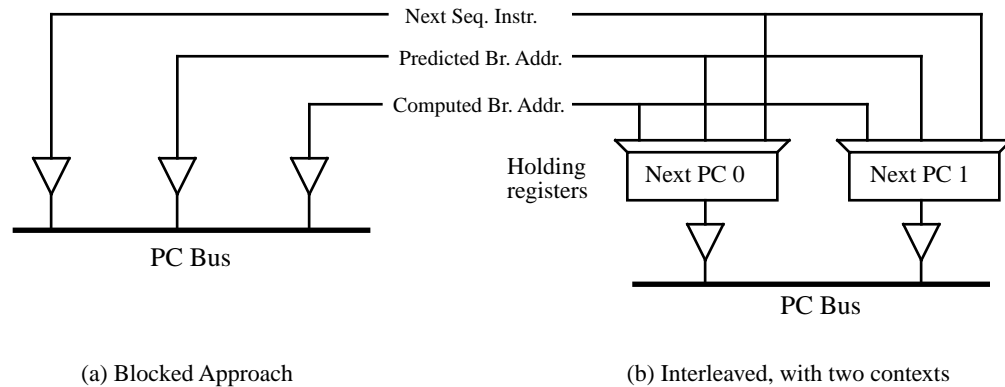| (a) Blocked Approach | (b) Interleaved, with two contexts |

Figure 11-34 Driving the PC bus in the blocked and interleaved multithreading approaches, with two contexts.

If more contexts were used, the interleaved scheme would require more replication, while the blocked scheme would not.

chain that runs along parallel to the PC chain, and is loaded and checked as the branch reaches the appropriate pipeline stages.

A final area of enhancement in the PC unit over a conventional processor is exception handling. Consider what happens when an exception occurs in one context. One choice is to have that context be rendered unready to make way for the exception handler, and let that handler be interleaved with the other user contexts (the Tera takes an approach similar to this). In this case, another user thread may also take an exception while the first exception handler is running, so the exception handlers must be able to cope with multiple concurrent handler executions. Another option is to render all the contexts unready when an exception occurs in any context, squash all the instructions in the pipeline, and reenable all contexts when the exception handler returns. This can cause a loss of performance if exceptions are frequent. It also means that the exception PCs (EPCs) of all active contexts need to be loaded with the first uncompleted instruction address from their respective threads when an exception occurs. This is more complicated than in the blocked case, where only the single EPC of the currently running and excepting context needs to be saved (see Exercise a).

Control

Two interesting issues related to control outside of the PC unit are tracking context availability information and feeding it to the PC unit, and choosing and switching to the next context every cycle. The context available signal is modified on a cache miss, when the miss returns, and on backoff instructions and expiration. Availability status for cache misses can be tracked by maintaining pending miss registers per context, which are loaded upon a miss and checked upon miss return to reenable the appropriate context. For explicit backoff instructions, we can maintain a counter per context, initialized to the backoff value when the backoff instruction is encountered (and the context availability signal cleared). The counter is decremented every cycle until it reaches zero, at which point the availability signal for that context is set again.

Backoff instructions can be used to tolerate instruction latency as well, but with the interleaving of contexts it may be difficult to choose a good number of backoff cycles. This is further complicated by the fact that the compiler may rearrange instructions transparently. Backoff values are implementation-specific, and may have to be changed for subsequent generations of processors.

Fortunately, short instruction latencies are often handled naturally by the interleaving of other contexts without any backoff instructions, as we saw in Figure 11-30. Robust solutions for long instruction latencies may require more complex hardware support such as scoreboarding.

As for choosing the next context, a reasonable approach once again is to choose round-robin qualified by context availability.

11.8.5 Integrating Multithreading with Multiple-Issue Processors

So far, our discussion of multithreading has been orthogonal to the number of operations issued per cycle. We saw that the Tera system issues three operations per cycle, but the packing of operations into wider instructions is done by the compiler, and the hardware chooses a three-operation instruction from a single thread in every cycle. A single thread usually does not have enough instruction-level parallelism to fill all the available slots in every cycle, especially if processors begin to issue many more operations per instruction. With many threads available anyway, an alternative is to let available operations from different threads be scheduled in the same cycle, thus filling instruction slots more effectively. This approach has been called simultaneous multithreading, and there have been many proposals for it [HKN+92, TEL95]. Another way of looking at it is that it is like interleaved multithreading, but operations from the different available threads compete for the functional units in every cycle.

Simultaneous multithreading seeks to overcome the drawbacks of both multiple-instruction issue and traditional (interleaved) multithreading by exploiting the simultaneous availability of operations from multiple threads. Simple multiple-issue processors suffer from two inefficiencies. First, not all slots in a given cycle are filled, due to limited ability to find instruction-level parallelism. Second, many cycles have nothing scheduled due to long-latency instructions. Simple multithreading addresses the second problem but not the first, while simultaneous multithreading tries to address both (see Figure 11-35).

(a) Single-threaded (a) Multi-threaded (Interleaved) (c) Simultaneous Multi-threaded
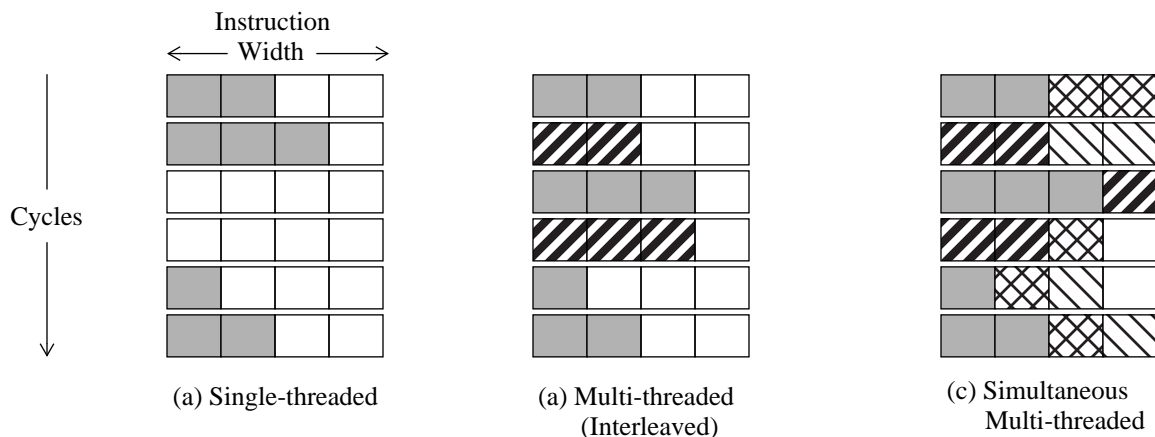
Figure 11-35 Simultaneous Multithreading.

The potential improvements are illustrated for both simple interleaved multithreading and simultaneous multithreading for a four-issue processor.

Choosing operations from different threads to schedule in the same cycle may be difficult for a compiler, but many of the mechanisms for it are already present in dynamically scheduled microprocessors. The instruction fetch unit must be extended to fetch operations from different hardware contexts in every cycle, but once operations from different threads are fetched and placed in the reorder buffer the issue logic can choose operations from this buffer regardless of which thread they are from. Studies of single-threaded dynamically scheduled processors have shown that the causes of empty cycles or empty slots are quite well distributed among instruction latencies, cache misses, TLB misses and load delay slots, with the first two often being particularly important. The variety of both the sources of wasted time and of their latencies indicates that fine-grained multithreading may be a good solution.

In addition to the issues for interleaved multiprocessors, several new issues arise in implementing simultaneously multithreaded processors [TEE+96]. First, in addition to the fundamental property of issuing instructions flexibly from different threads, the instruction fetch unit must be designed in a way that fetches instructions effectively from multiple active contexts as well. The more the flexibility allowed in fetching—compared to say fetching from only one context in a cycle or fetching at most two operations from each thread in a cycle—the more the complexity in the fetching logic and instruction cache design. However, more flexibility reduces the frequency of fetch slots being left empty because the thread that is allowed to fetch into those slots does not have enough ready instructions. Interesting questions also arise in choosing which context or contexts to fetch instructions from in the next cycle. We could choose contexts in a fixed order of priority (say try to fill from context 0 first, then fill the rest from context 1 and so on) or we could choose based on execution characteristics of the contexts (for example give priority to the context that has the fewest instructions currently in the fetch unit or the reorder buffer, or the context that has the fewest outstanding cache misses). Finally, we have to decide which operations to choose from the reorder buffer among the ones that are ready in each cycle. The standard practice in dynamically scheduled processors is to choose the oldest operation that is ready, but other choices may be more appropriate based on the thread the operations are from and how it is behaving.

There is little or no performance data on simultaneous multithreading in the context of multiprocessors. For uniprocessors, a performance study examines the potential performance benefits well as the impact of some of the tradeoffs discussed above [TEE+96]. That study finds, for example, that speculative execution is less important with simultaneous multithreading than with single-threaded dynamically scheduled processors, because there are more available threads and hence non-speculative instructions to choose from.

Overall, as data access and synchronization latencies become larger relative to processor speed, multithreading promises to become increasingly successful in hiding latency. Whether or not it will actually be incorporated in microprocessors depends on a host of other factors, such as what other latency tolerance techniques are employed (such as prefetching, dynamic scheduling, and relaxed consistency models) and how multithreading interacts with them. Since multithreading requires extra explicit threads and significant replication of state, an interesting alternative to it is to place multiple simpler processors on a chip and placing the multiple threads on different processors. How this compares with multithreading (e.g. simultaneous multithreading) in cost and performance is not yet well understood, for desktop systems or as a node for a larger multiprocessor.

This concludes our discussion of latency tolerance in a shared address space. Table 11-2 summa-

Table 11-2 Key properties of the three read-write latency hiding techniques in a shared address space.

Property	Relaxed Models	Prefetching	Multithreading	Block Data Transfer
Types of latency hidden	Write (blocking-read processors) Read (dynamically scheduled processors)	Write Read	Write Read Synchronization Instruction	Write Read
Requirements of software	Labeling synch. opera-tions	Predictability	Explicit extra concur-rency	Identifying and orches-trating block transfers
Hardware support	Little	Little	Substantial	Not in processor, but block transfer engine in memory system
Support in com-mercial micros?	Yes	Yes	Currently no	Not needed

rizes and compares some key features of the four major techniques used. The techniques can be and often are combined. For example, processors with blocking reads can use relaxed consistency models to hide write latency and prefetching or multithreading to hide read latency. And we have seen that dynamically scheduled processors can benefit from all of prefetching, relaxed consistency models and multithreading individually. How these different techniques interact in dynamically scheduled processors, and in fact how well prefetching might complement multithreading even in blocking-read processors, is likely to be better understood in the future.

11.9 Lockup-free Cache Design

Throughout this chapter, we have seen that in addition to the support needed in the processor— and the additional bandwidth and low occupancies needed in the memory and communication systems—several latency tolerance techniques in a shared address space require that the cache allow multiple outstanding misses if they are to be effective. Before we conclude this chapter, let us examine how such a lockup-free cache might be designed.

There are several key design questions for a cache subsystem that allows multiple outstanding misses:

- How many and what kinds of misses can be outstanding at the same time. Two distinct points in design complexity are: (i) a single read and multiple writes, and (ii) multiples reads and writes. Like the processor, it is easier for the cache to support multiple outstanding writes than reads.

- How do we keep track of the outstanding misses. For reads, we need to keep track of (i) the address of the word requested, (ii) the type of read request (i.e. read, read-exclusive, or prefetch, and single-word or double-word read), (iii) the place to return data when it comes into the cache (e.g. to which register within a processor, or to which processor if multiple processors are sharing a cache), and (iv) current status of the outstanding request. For writes, we do not need to track where to return the data, but the new data being returned must be merged with the data block returned by the next level of the memory hierarchy. A key issue here is whether to store most of this information within the cache blocks themselves or whether to

have a separate set of transaction buffers. Of course, while fulfilling the above requirements, we need to ensure that the design is deadlock/livelock free.

- How do we deal with conflicts among multiple outstanding references to the same memory block? What kinds of conflicting misses to a block should we allow and disallow (e.g., by stalling the processor)? For example, should we allow writes to words within a block to which a read miss is outstanding?

- How do we deal with conflicts between multiple outstanding requests that map to the same block in the cache, even though they refer to different memory blocks?

To illustrate the options, let us examine two different designs. They differ primarily in where they store the information that keeps track of outstanding misses. The first design uses a separate set of transaction buffers for tracking requests. The second design, to the extent possible, keeps track of outstanding requests in the cache blocks themselves.

The first design is a somewhat simplified version of that used in Control Data Corporation's Cyber 835 mainframe, introduced in 1979 [Kro81]. It adds a number of *miss state holding registers* (MSHRs) to the cache, together with some associated logic. Each MSHR handles one or more misses to a single memory block. This design allows considerable flexibility in the kinds of requests that can be simultaneously outstanding to a block, so a significant amount of state is stored in each MSHR as shown in Table 11-3.

Table 11-3 MSHR State Entries and their Roles.

State	Description	Purpose
Cache block pointer	Pointer to cache block allocated to this request	Which cache line within a set in set-associative cache?
Request address	Address of pending request	Allows merging of requests
Unit identification tags (one per word)	Identifies requesting processor unit	Where to return this word?
Send-to-cpu status (one per word)	Valid bits for unit identification tags	Is unit id tag valid?
Partial write codes (one per word)	Bit vector for tracking partial writes to a word	Which words returning from memory to overwrite
Valid indicator	Valid MSHR contents	Are contents valid?

The MSHRs are accessed in parallel to the regular cache. If the access hits in the cache, the normal cache hit actions take place. If the access misses in the cache, the actions depend on the content of the MSHRs:

- If no MSHR is allocated for that block, a new one is allocated and initialized. If none is free, or if all cache lines within a set have pending requests, then the processor stalls. If the allocated block contains dirty data, a writeback is initiated. If the processor request is a write, the data is written at the proper offset into the cache block, and the corresponding partial write code bits are set in the MSHR. A request to fetch the block from the main memory subsystem (e.g., BusRd, BusrdX) is also initiated.

- If an MSHR is already allocated for the block, the new request is merged with the previously pending requests for the same block. For example, a new write request can be merged by writing the data into the allocated cache block and by setting the corresponding partial write bits. A read request to a word that has completely been written in the cache (by earlier writes) can simply read the data from the cache memory. A read request to a word that has not been requested before is handled by setting the proper unit identification tags. If it is to a word that

has already been requested, then either a new MSHR must be allocated (since there is only one unit identification tag per word) or the processor must be stalled. Since a write does not need a unit identification tag, a write request for a word to which a read is already pending is handled easily: the data returned by main memory can simply be forwarded to the processor. Of course, the first such write to a block will have to generate a request that asks for exclusive ownership.

Finally, when the data for the block returns to the cache, the cache block pointer in the MSHR indicates where to put the contents. The partial write codes are used to avoid writing stale data into the cache, and the send-to-cpu bits and unit-identification-tags are used to forward replies to waiting functional units.

This design requires an associative lookup of MSHRs, but allows the cache to have all kinds of memory references outstanding at the same time. Fortunately, since the MSHRs do the complex task of merging requests and tearing apart replies for a given block, to the extended memory hierarchy below it appears simply as if there are multiple requests to distinct blocks coming from the processor. The coherence protocols we have discussed in the previous chapters are already designed to multiple outstanding requests to different blocks without deadlock.

The alternative to this design is to store most of the relevant state in the cache blocks themselves, and not use separate MSHRs. In addition to the standard MESI states for a writeback cache, we add three transient or pending states: *invalid-pending* (IP), *shared-pending* (SP), and *exclusive-pending* (EP), corresponding to what the state of the block was when the write was issued. In each of these three states, the cache tag is valid and the cache block is awaiting data from the memory system. Each cache block also has *subblock write bits* (SWBs), which is a bit-vector with one bit per word. In IP state, all SWBs must be OFF and all words are considered invalid. In both EP and SP states, the bits that are turned ON indicate the words in the block that have been written by the processor since the block was requested from memory, and which the data returned from memory should not overwrite. However, in the EP state, words for which the bits are OFF are considered invalid, while in the SP state, those words are considered valid (not stale). Finally, there is a set of pending-read registers; these contain the address and type of pending read requests.

The key benefit of keeping this extra state information with each cache block is that no additional storage is needed to keep track of pending write requests. On a write that does not find the block in modified state, the block simply goes into the appropriate pending state (depending on what state it was in before), initiates the appropriate bus transaction, and sets the SWB bits to indicate which words the current write has modified so the subsequent merge will happen correctly. Writes that find the block already in pending state only require that the word is written into the line and the corresponding SWB is set (this is true except if the state was invalid-pending—i.e. a request for a shared block is outstanding—in which case the state is changed to EP and a read-exclusive request is generated).

Reads may use the pending-read registers. If a read finds the desired word in a valid state in the block (including a pending state with the SWB on), then it simply returns it. Otherwise, it is placed in a pending-read register which keeps track of it. Of course, if the block accessed on a read or write is not in the cache (no tag match) then a writeback may be generated, the block is set to the invalid-pending state, all SWBs are turned off (except of the word being written if it is a write) and the appropriate transaction is placed on the bus. If the tag does not match and the existing block is already in pending state, then the processor stalls. Finally, when a response to an out-

standing request arrives, the corresponding cache block is updated except for the words that have their SWBs on. The cache block moves out of the pending state. All pending read-registers are checked to see if any were waiting for this cache block; if so, data is returned for those requests and those pending-read registers freed. Details of the actual state changes, actions and race conditions can be found in [Lau94]. One key observation that makes race conditions relatively easy to deal with is that even though words are written into cache blocks before ownership is obtained for the block, those words are not visible to requests from other processors until ownership is obtained.

Overall, the two designs described above are not that different conceptually. The latter solution keeps the state for writes in the cache blocks and does not use separate tracking registers for writes. This reduces the number of pending registers needed and the complexity of the associative lookups, but it is more memory intensive since extra state is stored with all lines in the cache even though only a few of them will have an outstanding request at a time. The correctness interactions with the rest of the protocol are similar and modest in the two cases.

11.10 Concluding Remarks

With the increasing gap between processor speeds and memory access and communication times, latency tolerance is likely to be critical in future multiprocessors. Many latency tolerance techniques have been proposed and are used, and each has its own relative advantages and disadvantages. They all rely on excess concurrency in the application program beyond the number of processors used, and they all tend to increase the bandwidth demands placed on the communication architecture. This greater stress makes it all the more important that the different performance aspects of the communication architecture (the processor overhead, the assist occupancy and the network bandwidth) be both efficient and also well balanced.

For cache-coherent multiprocessors, latency tolerance techniques are supported in hardware by both the processor and the memory system, leading to a rich space of design alternatives. Most of these hardware-supported latency tolerance techniques are also applicable to uniprocessors, and in fact their commercial success depends on their viability in the high-volume uniprocessor market where the latencies to be hidden are smaller. Techniques like dynamic scheduling, relaxed memory consistency models and prefetching are commonly encountered in microprocessor architectures today. The most general latency hiding technique, multithreading, is not yet popular commercially, although recent directions in integrating multithreading with dynamically scheduled superscalar processors appear promising.

Despite the rich space of issues in hardware support, much of the latency tolerance problem today is a software problem. To what extent can a compiler automate prefetching so a user does not have to worry about it? And if not, then how can the user naturally convey information about what and when to prefetch to the compiler? If block transfer is indeed useful on cache-coherent machines, how will users program to this mixed model of implicit communication through reads and writes and explicit transfers? Relaxed consistency models carry with them the software problem of specifying the appropriate constraints on reordering. And will programs be decomposed and assigned with enough extra explicit parallelism that multithreading will be successful? Automating and simplifying the software support required for latency tolerance is a task that is far form fully accomplished. In fact, how latency tolerance techniques will play out in the future and what software support they will use is an interesting open question in parallel architecture.

11.11 References

[AHA+97]   Hazim A. Abdel-Shafi, Jonathan Hall, Sarita V. Adve and Vikram S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In Proceedings of the Third Symposium on High Performance Computer Architecture, February 1997.

[ALK+91]   Agarwal, A., Lim, B.-H., Kranz, D. and Kubiatowicz, J. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990, pp. 104-114.

[ABC+95]   Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 2-13, June 1995.

[ACC+90]   Alverson, R., Callahan, D., Cummings, D., Koblenz, B, Porterfield, A., and Smith, B. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, June 1990, pp. 1-6.

[BaC91]   Jean-Loup Baer and Tien-Fu Chen. An Efficient On-chip Preloading Scheme to Reduce Data Access Penalty. In Proceedings of Supercomputing'91, pp. 176-186, November 1991.

[BeF96a]   Bennett, J.E. and Flynn, M.J. Latency Tolerance for Dynamic Processors. Technical Report no. CSL-TR-96-687, Computer Systems Laboratory, Stanford University, 1996.

[BeF96b]   Bennett, J.E. and Flynn, M.J. Reducing Cache Miss Rates Using Prediction Caches. Technical Report no. CSL-TR-96-707, Computer Systems Laboratory, Stanford University, 1996.

[ChB92]   Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-Blocking and Prefetching Caches. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 51-61, 1992.

[ChB94]   Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In Proceedings of the 21st Annual Symposium on Computer Architecture, pp. 223-232, April 1994.

[DDS95]   Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, vol. 6, no. 7, July 1995.

[DFK+92]   William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes and Peter R. Nuth. The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. IEEE Micro, April 1992, pp. 23-39.

[FuP91]   John W.C. Fu and Janak H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In Proceedings of the 18th Annual Symposium on Computer Architecture, pp. 54-63. May 1991.

[FPJ92]   John W. C. Fu and Janak H. Patel and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 102-110, December 1992.

[GGH91a]   Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 245-257, April 1991.

[GGH91b]   Gharachorloo, K., Gupta, A., and Hennessy, J. Two Techniques to Enhance the Performance of Memory Consistency Models. In Proceedings of the International Conference on Parallel Processing, 1991, pp. I355-I364.

[GGH92] Gharachorloo, K., Gupta, A. and Hennessy, J. Hiding Memory Latency Using Dynamically Scheduling in Shared Memory Multiprocessors. In Proceedings of the 19th International Symposium on Computer Architecture, 1992, pp. 22-33.

[GrS96] Hekan Grahn and Per Stenstrom. Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection. Journal of Parallel and Distributed Computing, vol. 39, no. 2, pp. 168-180, December 1996.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In Proceedings of the 18th International Symposium on Computer Architecture, pp. 254-263, May 1991.

[HGD+94] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 38-50, October 1994.

[HBG+97] John Heinlein, Robert P. Bosch, Jr., Kourosh Gharachorloo, Mendel Rosenblum, and Anoop Gupta. Coherent Block Data Transfer in the FLASH Multiprocessor. In Proceedings of the 11th International Parallel Processing Symposium, April 1997.

[HeP95] Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Second Edition, Morgan Kaufman, 1995.

[HKN+92] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase and Teiji Nishizawa. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In Proceedings of the 19th International Symposium on Computer Architecture, pp. 136-145, May 1992.

[Hun96] Hunt, D. Advanced features of the 64-bit PA-8000. Hewlett Packard Corporation, 1996.

[Int96] Intel Corporation. Pentium(r) Pro Family Developer's Manual.

[Jor85] Jordan, H. F. HEP Architecture, Programming, and Performance. In *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, J S. Kowalik, ed., MIT Press, 1985, page 8.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In Proceedings of the 17th Annual Symposium on Computer Architecture, pp. 364-373, June 1990.

[Kro81] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pp. 81-87, May 1981.

[KuA93] John Kubiatowicz and Anant Agarwal. The Anatomy of a Message in the Alewife Multiprocessor. In Proceedings of the International Conference on Supercomputing, pp. 195-206, July 1993.

[KuS88] Kuehn, J.T, and Smith, B.J. The Horizon Supercomputing System: Architecture and Software. In *Proceedings of Supercomputing'88*, November 1988, pp. 28-34.

[KCA91] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. Proceedings of the International Symposium on Shared Memory Multiprocessing, pp. 91-101, April 1991.

[Lau94] Laudon, J. Architectural and Implementation Tradeoffs in Multiple-context Processors. Ph.D. Thesis, Stanford University, Stanford, California, 1994. Also published as Technical Report CSL-TR-94-634, Computer Systems Laboratory, Stanford University, May 1994.

[LGH94] Laudon, J., Gupta, A. and Horowitz, M. Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, edited by R.A. Iannucci, Kluwer Academic Publishers, 1994, pp. 167-200.

[LKB91] R. L. Lee, A. Y. Kwok, and F. A. Briggs. The Floating Point Performance of a Superscalar

SPARC Processor. In Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems, April. 1991.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[LEE+97] J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen. Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading. ACM Transactions on Computer Systems, vol. xx, no. xx, August, 1997).

[LuM96] Luk, C.-K. and Mowry, T.C. Compiler-based Prefetching for Recursive Data Structures. In Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), October 1996.

[MIP96] MIPS Technologies, Inc. R10000 Microprocessor User's Manual, Version 1.1, January 1996.

[Mow94] Todd C. Mowry. Tolerating Latency through Software-Controlled Data Prefetching. Ph. D. Thesis, Computer Systems Laboratory, Stanford University, 1994. Also Technical Report no. CSL-TR-94-628, Computer Systems Laboratory, Stanford University, June 1994.

[NWD93] Michael D. Noakes and Deborah A. Wallach and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In Proceedings of the 20th International Symposium on Computer Architecture, pp. 224-235, May 1993.

[NuD95] Nuth, Peter R. and Dally, William J. The Named-State Register File: Implementation and Performance. In Proceedings of the First International Symposium on High-Performance Computer Architecture, January 1995.

[Oha96] Moriyoshi Ohara. Producer-Oriented versus Consumer-Oriented Prefetching: a Comparison and Analysis of Parallel Application Programs. Ph. D. Thesis, Computer Systems Laboratory, Stanford University, 1996. Available as Stanford University Technical Report No. CSL-TR-96-695, June 1996.

[Omo94] Amos R. Omondi. Ideas for the Design of Multithreaded Pipelines. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, Robert Iannucci, Ed., Kluwer Academic Publishers, 1994. See also Amos R. Omondi, Design of a high performance instruction pipeline, Computer Systems Science and Engineering, vol. 6, no. 1, January 1991, pp. 13-29.

[PRA+96] Pai, V.S., Ranganathan, P., Adve, S, and Harton, T. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), October 1996.

[RPA+97] Ranganathan, P., Pai, V.S., Abdel-Shafi, H. and Adve, S.V. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[SWG92] Singh, J.P., Weber, W-D., and Gupta, A. SPLASH: The Stanford ParalleL Applications for Shared Memory. Computer Architecture News, vol. 20, no. 1, pp. 5-44, March 1992.

[Sin97] Jaswinder Pal Singh. Some Aspects of Controlling Scheduling in Hardware Control Prefetching. Technical Report no. xxx, Computer Science Department, Princeton University. Also available on the World Wide Web home page of *Parallel Comptuer Architecture* by Culler and Singh published by Morgan Kaufman publishers.

[Smi81] Smith, B.J. Architecture and Applications of the HEP Multiprocessor Computer System. In Proceedings of SPIE: Real-Time Signal Processing IV, Vol. 298, August, 1981, pp. 241-248.

[Smi85] Smith, B.J. The Architecture of HEP. In *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, edited by J.S. Kowalik, MIT Press, 1985, pp. 41-55.

[Tom67] Tomasulo, R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal of Research and Development, 11:1, January 1967, pp. 25-33.

[TuE93] Dean M. Tullsen and Susan J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multi-processor. In Proceedings of the 20th Annual Symposium on Computer Architecture, pp. 278-288, May 1993.

[TEL95] Dean M. Tullsen, Susan J. Eggers and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In Proceedings of the 20th Annual Symposium on Computer Architecture, pp. 392-403, June 1995.

[TEE+96] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo and R.L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In Proceedings of the 23rd International Symposium on Computer Architecture, May 1996.

[WoL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp. 30-44, June 1991.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219-229, San Jose, CA, October 1994.

[ZhT95] Zheng Zhang and Josep Torrellas. Speeding up Irregular Applications in Shared-Memory Multi-processors: Memory Binding and Group Prefetching. In Proceedings of the 22nd Annual Symposium on Computer Architecture, pp. 188-199, May 1995.

11.12 Exercises

11.1 General.

    a. Why is latency reduction generally a better idea than latency tolerance?

    b. Suppose a processor communicates k words in m messages of equal size, the assist occupancy for processing a message is o, and there is no overhead on the processor. What is the best case latency as seen by the processor if only communication, not computation, can be overlapped with communication? First, assume that acknowledgments are free; i.e. are propagated instantaneously and don't incur overhead; and then include acknowledgments. Draw timelines, and state any important assumptions.

    c. You have learned about a variety of different techniques to tolerate and hide latency in shared-memory multiprocessors. These techniques include blocking, prefetching, multiple context processors, and relaxed consistency models. For each of the following scenarios, discuss why each technique will or will not be an effective means of reducing/hiding latency. List any assumptions that you make.

    (i) A complex graph algorithm with abundant concurrency using linked pointer structures.

    (ii) A parallel sorting algorithm where communication is producer initiated and is achieved through long latency write operations. Receiver-initiated communication is not possible.

    (iii) An iterative system-of-equation solver. The inner loop consists of a matrix-matrix multiply. Assume both matrices are huge and don't fit into the cache.

11.2 You are charged with implementing message passing on a new parallel supercomputer. The architecture of the machine is still unsettled, and your boss says the decision of whether to

provide hardware cache coherence will depend on the message passing performance of the two systems under consideration, since you want to be able to run message passing applications from the previous-generation architecture.

In the system without cache coherence (the "D" system), the engineers on your team tell you that message passing should be implemented as successive transfers of 1KB. To avoid problems with buffering at the receiver side, you're required to acknowledge each individual 1KB transfer before the transmission of the next one can begin (so, only one block can be in flight at a time). Each 1KB line requires 200 cycles of setup time, after which it begins flowing into the network. This overhead accounts for time to determine where to read the buffer from in memory and to set up the DMA engine which performs the transfer. Assume that from the time that the 1KB chunk reaches the destination, it takes 20 cycles for the destination node to generate a response, and it takes 50 cycles on the sending node to accept the ACK and proceed to next 1KB transfer.

In the system with cache coherence (the "CC" system), messages are sent as a series of 128 byte cache line transfers. In this case, however, acknowledgments only need to be sent at the end of every 4KB page. Here, each transfer requires 50 cycles of setup time, during which time the line can be extracted from the cache, if necessary, to maintain cache coherence. This line is then injected into the network, and only when the line is completely injected into the network can processing on the next line begin.

The following are the system parameters: Clock rate = 10 ns (100 MHz), Network latency = 30 cycles, Network bandwidth = 400 MB/s. State any other assumptions that you make.

a. What is the latency (until the last byte of the message is received at the destination) and sustainable bandwidth for a 4KB message in the D system?

b. What is the corresponding latency and bandwidth in the CC system?

c. A designer on the team shows you that you can easily change the CC system so that the processing for the next line occurs while the previous one is being injected into the network. Calculate the 4KB message latency for the CC system with this modification.

11.3 Consider the example of transposing a matrix of data in parallel, as is used in computations such as high-performance Fast Fourier Transforms. Figure 11-36 shows the transpose pictorially. Every processor tranposes one "patch" of its assigned rows to every other processor, including one to itself. Performing the tranpose through reads and writes was discussed in the exercises at the end of Chapter 8. Since it is completely predictable which data a processor has to send to which other processors, a processor can send an entire patch at a time in a single message rather than communicate the patches through individual read or write cache misses.

a. What would you expect the curve of block transfer performance relative to read-write performance to look like?

b. What special features would the block transfer engine benefit most from?

c. Write pseudocode for the block transfer version of the code.

d. Suppose you wanted to use block data transfer in the Raytrace application. For what purposes would you use it, and how? Do you think you would gain significant performance benefits?

11.4 An interesting performance issue in block data transfer in a cache-coherent shared address space has to do with the impact of long cache blocks. Assume that the data movement in the block transfer leverages the cache coherence mechanisms. Consider the simple equation
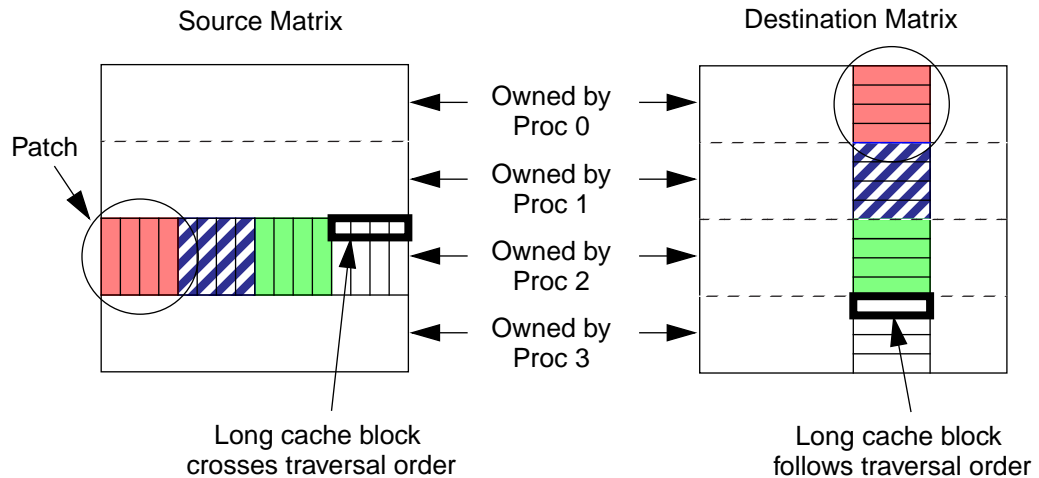
Figure 11-36 Sender-initiated matrix transposition.

The source and destination matrices are partitioning among processes in groups of contiguous rows. Each process divides its set of n/p rows into p patches of size n/p * n/p. Consider process P2 as a representative example: It sends one patch to every other process, and transposes one patch (third from left, in this case) locally. Every patch may be transferred as a single block transfer message, rather than through individual remote writes or remote reads (if receiver-initiated).

solver on a regular grid, with its near neighbor communication. Suppose the n-by-n grid is partitioned into square subblocks among p processors.

a. Compared to the results shown for FFT in the chapter, how would you expect the curves for this application to differ when each row or column boundary is sent directly in a single block transfer and why?

b. How might you structure the block transfer to only send useful data, and how would you expect performance in this case to compare with the previous one?

c. What parameters of the architecture would most affect the tradeoffs in b.?

d. If you indeed wanted to use block transfer, what deeper changes might you make to the parallel program?

11.5 Memory Consistency Models

a. To maintain all program orders as in SC, we can optimize in the following way. In our baseline implementation, the processor stalls immediately upon a write until the write completes. The alternative is to place the write in the write buffer without stalling the processor. To preserve the program order among writes, what the write buffer does is retire a write (i.e. pass it further along the memory hierarchy and make it visible to other processors) only after the write is complete. The order from writes to reads is maintained by flushing the write buffer upon a read miss.

(i) What overlap does this optimization provide?

(ii) Would you expect it to yield a large performance improvement? Why or why not?

b. If the second-level cache is blocking in the performance study of proceeding past writes with blocking reads, is there any advantage to allowing write->write reordering over not allowing it? If so, under what conditions?

c. Write buffers can allow several optimizations, such as *buffering* itself, *merging* writes to the same cache line that is in the buffer, and forwarding values from the buffer to read operations that find a match in the buffer. What constraints must be imposed on these optimizations to maintaining program order under SC? Is there a danger with the merging optimizations for the processor consistency model?

11.6 Give a bunch of code sequences and a architecture latency model, calculate completion time under various memory consistency models. Plenty of questions in cs315a exams of this sort.

11.7 Prefetching

a. How early can you issue a binding prefetch in a cache-coherent system?

b. We have talked about the advantages of nonbinding prefetches in multiprocessors. Do nonbinding prefetches have any advantages in uniprocessors as well. Assume that the alternative is to prefetch directly into the register file, not into a prefetch buffer.

c. Sometimes a predicate must be evaluated to determine whether or not to issue a prefetch. This introduces a conditional (if) expression around the prefetch inside the loop containing the prefetches. Construct an example, with pseudocode, and describe what is the performance problem? How would you fix the problem?

d. Describe situations in which a producer-initiated deliver operation might be more appropriate than prefetching or an update protocol. Would you implement a deliver instruction if you were designing a machine?

11.8 Prefetching Irregular Accesses

a. Consider the loop

```
for i <- 1 to 200
     sum = sum + A[index[i]];
end for.
```

Write a version of the code with non-binding software-controlled prefetches inserted. Include the prologue, the steady state of the software pipeline, and the epilogue. Assume the memory latency is such that it takes five iterations of the loop for a data item to return, and that data are prefetched into the primary cache.

b. When prefetching indirect references such as in the example above, extra instructions are needed for address generation. One possibility is to save the computed address in a register at the time of the prefetch and then reuse it later for the load? Are there any problems with or disadvantages to this?

c. What if an exception occurs when prefetching down multiple levels of indirection in accesses. What complications are caused and how might they be addressed?

d. Describe some hardware mechanisms (at a high level) that might be able to prefetch iregular accesses, such as records or lists.

11.9 More prefetching

a. Show how the following loop would be rewritten with prefetching so as to hide latency:

```
for i=0 to 128 {
   for j=1 to 32
      A[i,j] = B[i] * C[j]
}
```

Try to reduce overhead by prefetching only those references that you expect to miss in the cache. Assume that a read prefetch is expressed as "PREFETCH(&variable)", and it fetches the entire cache line in which `variable` resides in shared mode. A read-exclu-

sive prefetch operation which is expressed as "RE_PREFETCH(&variable)" which fetches the line in "exclusive" mode. The machine has a cache miss latency of 32 cycles. Explicitly prefetch each needed variable (don't take into account the cache block size). Assume that the cache is large (so you don't have to worry about interference), but not so large that you can prefetch everything at the start. In other words, we are looking for just-in-time prefetching. The matrix A[i,j] is stored in memory with A[i,j] and A[i,j+1] contiguous. Assume that the main computation of the loop takes 8 cycles to complete.

b. Construct an example in which a prefetching compiler's decision to assume everything in the cache is invalidated when it sees a synchronization operation is very conservative, and show how a programmer can do better. It might be useful to think of the case study applications used in the book.

c. One alternative to prefetching is to use nonblocking load operations, and issue these operations significantly before the data are needed for computation. What are the tradeoffs between prefetching and using nonblocking loads in this way?

d. There are many options for the format that a prefetch instruction can take. For example, some architectures allow a load instruction to have multiple flavors, one of which can be reserved for a prefetch. Or in architectures that reserve a register to always have the value zero (e.g. the MIPS and SPARC architectures); a load with that register as the destination can be interpreted as a prefetch since such a load does not change the contents of the register. A third option is to have a separate prefetch instruction in the instruction set, with a different opcode than a load.

(i) Which do you think is the best alternative and why?

(ii) What addressing mode would you use for a prefetch instruction and why?

e. Suppose we issue prefetches when we expect the corresponding references to miss in the primary (first-level) cache. A question that arises is which levels of the memory hierarchy beyond the first-level cache should we probe to see if the prefetch can be satisfied there. Since the compiler algorithm usually schedules prefetches by conservatively assuming that the latency to be hidden is the largest latency (uncontended, say) in the machine, one possibility is to not even check intermediate levels of the cache hierarchy but always get the data from main memory or from another processor's cache if the block is dirty. What are the problems with this method and which one do you think is most important?

f. We have discussed some qualitative tradeoffs between prefetching into the primary cache and prefetching only into the second-level cache. Using only the following parameters, construct an analytical expression for the circumstances under which prefetching into the primary cache is beneficial. What about the analysis or what it leaves out strikes you as making it most difficult to rely on it in practice? p_t is the number of prefetches that bring data into the primary cache early enough, p_d is the number of cases in which the prefetched data are displaced from the cache before they are used, p_c is the number of cache conflicts in which prefetches replace useful data, p_f is the number of prefetch fills (i.e. the number of times a prefetch tries to put data into the primary cache), l_s is the access latency to the second-level cache (beyond that to the primary cache) and l_f is the average number of cycles that a prefetch fill stalls the processor. After generating the full-blown general expression, your goal is to find a condition on l_f that makes prefetching into the first level cache worthwhile. To do this, you can make the following simplifying assumptions: $p_s = p_t - p_d$, and $p_c = p_d$.

11.10 Multithreading. Consider a "blocked" context-switching processor. (i.e. a processor that switches contexts only on long-latency events.) Assume that there are arbitrarily many threads available and clearly state any other assumptions that you make in answering the fol-

lowing questions. The threads of a given application have been analyzed to show the following execution profile:

> 40% of cycles are spent on instruction execution (busy cycles)

> 30% of cycles spent stalled on L1 cache misses but L2 hits (10 cycle miss penalty)

> 30% of cycles spent stalled on L2 cache misses (30 cycle miss penalty)

a. What will be the busy time if the context switch latency (cost) is 5 cycles?

b. What is the maximum context switch latency that will ensure that busy time is greater than or equal to 50%?

11.11   Blocked multithreading

a. In blocked, multiple-context processors with caches, a context switch occurs whenever a reference misses in the cache. The blocking context at this point goes into "stalled" state, and it remains there until the requested data arrives back at the cache. At that point it returns to "active" state, and it will be allowed to run when the active contexts ahead of it block. When an active context first starts to run, it reissues the reference it had blocked on. In the scheme just described, the interaction between the multiple contexts can potentially lead to deadlock. Concretely describe an example where none of the contexts make forward process.

b. What do you think would happen to the idealized curve for processor utilization versus degree of multithreading in Figure 11-25 if cache misses were taken into account. Draw this more realistic curve on the same figure as the idealized curve.

c. Write the logic equation that decides whether to generate a context switch in the blocked scheme, given the following input signals: Cache Miss, MissSwitchEnable (enable switching on a cache miss), CE (signal that allows processor to enable context switching), OneCount(CValid) (number of ready contexts), ES (explicit context switch instruction), and TO (timeout). Write another equation to decide when the processor should stall rather than switch.

d. In discussing the implementation of the PC unit for the blocked multithreading approach, we said that the use of the exception PC for exceptions as well as context switches meant that context switching must be disabled upon an exception. Does this indicate that the kernel cannot use the hardware-provided multithreading at all? If so, why? If not, how would you arrange for the kernel to use the multiple hardware contexts.

11.12   Interleaved Multithreading

a. Why is exception handling more complex in the interleaved scheme than in the blocked scheme? [Hint: think about how the exception PC (EPC) is used.] How would you handle the issues that arise? :

b. How do you think the Tera processor might do lookahead across branches? The jprocessor provides JUMP_OFTEN and JUMP_SELDOM branch operations. Why do you think it does this?

c. Consider a simple, HEP-like multithreaded machine with no caches. Assume that the average memory latency is 100 clock cycles. Each context has blocking loads and the machine enforces sequential consistency.

> (i) Given that 20% of a typical workload's instructions are loads and 10% are stores, how many active contexts are needed to hide the latency of the memory operations?

> (ii) How many contexts would be required if the machine supported release consistency (still with blocking loads)? State any assumptions that you make.

(iii) How many contexts would be needed for parts (i) and (ii) if we assumed a blocked multiple-context processor instead of the cycle-by-cycle interleaved HEP processor. Assume cache hit rates of 90% for both loads and stores.

(iv) For part (iii), what is the peak processor utilization assuming a context switch overhead of 10 cycles.

11.13 Studies of applications have shown that combining release consistency and prefetching always results in better performance than when either technique is used alone. This is not the case when multiple-contexts and prefetching techniques are combined; the combined performance can sometimes be worse. Explain the latter observation, using an example situation to illustrate.

CHAPTER 12   Future Directions

In the course of writing this book the single factor that stood out most among the many interesting facets of parallel computer architecture was the tremendous pace of change. Critically important designs became "old news" as they were replace by newer designs. Major open questions were answered, while new ones took their place. Start-up companies left the marketplace as established companies made bold strides into parallel computing and powerful competitors joined forces. The first teraflops performance was achieved and workshops had already been formed to understand how to accelerate progress towards petaflops. The movie industry produced its first full-length computer animated motion picture on a large cluster and for the first time a parallel chess program defeated a grand master. Meanwhile, multiprocessors emerged in huge volume with the Intel Pentium Pro and its glueless cache coherence memory bus. Parallel algorithms were put to work to improve uniprocessor performance by better utilizing the storage hierarchy. Networking technology, memory technology, and even processor design were all thrown "up for grabs" as we began looking seriously at what to do with a billion transistors on a chip.

Looking forward to the future of parallel computer architecture, the one prediction that can be made with certainty is continued change. The incredible pace of change makes parallel computer architecture an exciting field to study and in which to conduct research. One needs to continually revisit basic questions, such as what are the proper building blocks for parallel machines? What are the essential requirements on the processor design, the communication assist and how it integrates with the processor, the memory and the interconnect? Will these continue to utilize commodity desktop components, or will there be a new divergence as parallel computing matures and the great volume of computers shifts into everyday appliances? The pace of change makes for rich opportunities in industry, but also for great challenges as one tries to stay ahead.

Although it is impossible to predict precisely where the field will go, this final chapter seeks to outline some of the key areas of development in parallel computer architecture and the related technologies. Whatever directions the market takes and whatever technological breakthroughs occur, the fundamental issues addressed throughout this book will still apply. The realization of parallel programming models will still rest upon the support for naming, ordering, and synchronization. Designers will still battle with latency, bandwidth, and cost. The core techniques for addressing these issue will remain valid, however, the way that they are employed will surely change as the critical coefficients of performance, cost, capacity, and scale continue to change. New algorithms will be invented, changing the fundamental application workload requirements, but the basic analysis techniques will remain.

Given the approach taken throughout the book, it only makes sense of structure the discussion of potential future directions around hardware and software. For each, we need to ask what trends are likely to continue, thus providing a basis for evolutionary development, and which are likely stop abruptly, either because a fundamental limit is struck or because a breakthrough changes the direction. Section 12.1 examines trends in technology and architecture, while Section 12.2 looks at how changing software requirements may influence the direction of system design and considers how the application base is likely to broaden and change.

12.1 Technology and Architecture

Technological forces shaping the future of parallel computer architecture can be placed into three categories: evolutionary forces, as indicated by past and current trends, fundamental limits that wall off further progress along a trend, and breakthroughs that create a discontinuity and establish new trends. Of course, only time will tell how these actually play out. This section examines all three scenarios and the architectural changes that might arise.

To help sharpen the discussion, let us consider two questions. At the high end, how will the next factor of 1000 increase in performance be achieved? At the more moderate scale, how will cost effective parallel systems evolve? In 1997 computer systems form a parallelism pyramid roughly as in Figure 12-1. Overall shipments of uniprocessor PCs, workstations, and servers is on the order of tens of millions. The 2-4 processor end of the parallel computer market is on the scale of 100K to 1 M. These are almost exclusively servers, with some growth toward the desktop. This segment of the market grew at a moderate pace throughout the 80s and early 90's and then shot up with the introduction of Intel based SMPs manufactured by leading PC vendors, as well as the traditional workstation and server vendors are pushing down to expand volume. The next level is occupied by machines of 5 to 30 processors. These are exclusively high-end servers. The volume is in the tens of thousands of units and has been growing steadily; this segment dominates the

high-end server market, including the Enterprise market, which used to be called the mainframe market. At the scale of several tens to a hundred processors, the volume is on order a thousand systems. These tend to be dedicated engines supporting massive data bases, large scientific applications, or major engineering investigations, such as oil exploration, structural modeling, or fluid dynamics. Volume shrinks rapidly beyond a hundred processors, with order tens of systems at the thousand processor scale. Machines at the very top end have been on the scale of a thousand to two thousand processors since 1990. In 1996-7 this stepped up toward ten thousand processors. The most visible machines at the very top end being dedicated to advanced scientific computing, including the U.S. Department of Energy "ASCI" teraflops machines and the Hitachi SR2201 funded by the Japanese Ministry of Technology and Industry.
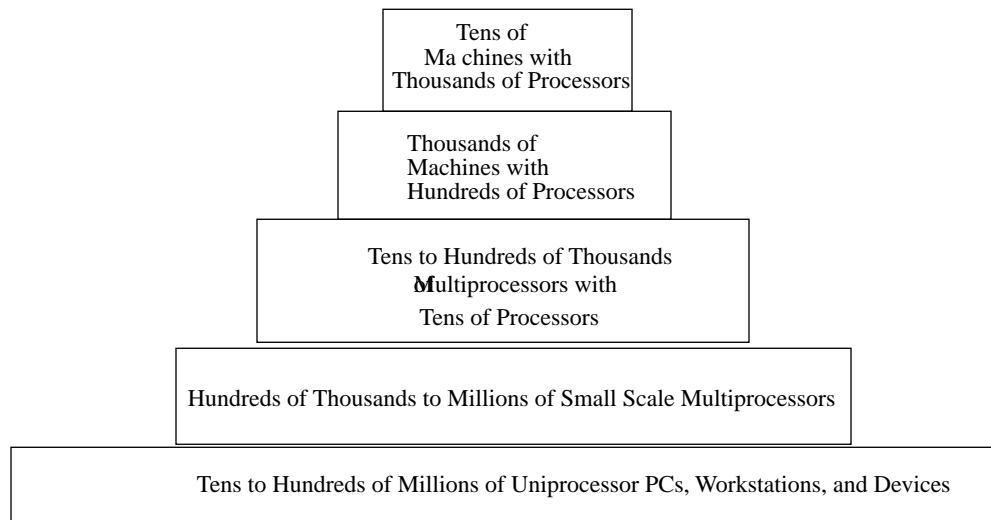

Figure 12-1 Market Pyramid for Parallel Computers

12.1.1 Evolutionary Scenario

If current technology trends hold, parallel computer architecture can be expected to follow an evolutionary path, in which economic and market forces play a crucial role. Let's expand this evolutionary forecast, then we can consider how the advance of the field may diverge from this path. Currently, we see processor performance increasing by about a factor of 100 per decade (or 200 per decade if the basis is Linpack or SpecFP). DRAM capacity also increases by about a factor of 100 per decade (quadrupling every three years). Thus, current trends would suggest that the basic balance of computing performance to storage capacity (MFlops/MB) of the nodes in parallel machines could remain roughly constant. This ratio varies considerably in current machines, depending on application target and cost point, but under the evolutionary scenario the family of options would be expected continue into the future with dramatic increases in both capacity and performance. Simply riding the commodity growth curve, we could look toward achieving petaflops scale performance by 2010, or perhaps a couple of years earlier if the scale of parallelism is increased, but such systems would be in excess of 100 million dollars to construct. It is less clear what communication performance these machines will provide, for reasons that are discussed below. To achieve this scale of performance a lot earlier in a general purpose system, would

involve an investment and a scale of engineering that is probably not practical, although special purpose designs may push up the time frame for a limited set of applications.

To understand what architectural directions may be adopted, the VLSI technology trends underlying the performance and capacity trends of the components are important. Microprocessor clock rates are increasing by a factor of 10-15 per decade, while transistors per microprocessor increase by more than a factor of 30 per decade. DRAM cycles times, on the other hand, improve much more slowly, roughly a factor of two per decade. Thus, the gap between processor speed and memory speed is likely to continue to widen. In order to stay on the processor performance growth trend, the increase in the ratio of memory access time to processor cycle time will require that processors employ better latency avoidance and latency tolerance techniques. Also, the increase in processor instruction rates, due to the combination of cycle time and parallelism, will demand that the bandwidth delivered by the memory increase.[1]

Both of these factors, the need for latency avoidance and for high memory bandwidth, as well as the increase in on-chip storage capacity, will cause the storage hierarchy to continue to become deeper and more complex. Recall, caches reduce the sensitivity to memory latency by avoiding memory references and they reduce the bandwidth required of the lower level of the storage hierarchy. These two factors will also cause the degree dynamic scheduling of instruction level parallelism to increase. Latency tolerance fundamentally involves allowing a large number of instructions, including several memory operations, to be in progress concurrently. Providing the memory system with multiple operations to work on at a time allows pipelining and interleaving to be used to increase bandwidth. Thus, the VLSI technology trends are likely to encourage the design of processors that are both more insulated from the memory system and more flexible, so that they can adapt to the behavior of the memory system. This bodes well for parallel computer architecture, because the processor component is likely to become increasingly robust to infrequent long latency operations.

Unfortunately, neither caches nor dynamic instruction scheduling reduce the actual latency on an operation that crosses the processor chip boundary. Historically, each level of cache added to the storage hierarchy increases the cost of access to memory. (For example, we saw that the Cray T3D and T3E designers eliminated a level of cache in the workstation design to decrease the memory latency, and the presence of a second level on-chip cache in the T3E increased the communication latency.) This phenomenon of increasing latency with hierarchy depth is natural because designers rely on the hit being the frequent case; increasing the hit rate and reducing the hit cost do more for processor performance than decreasing the miss cost. The trend toward deeper hierarchies presents a problem for parallel architecture, since communication, by its very nature, involves crossing out of the lowest level of the memory hierarchy on the node. The miss penalty contributes to the communication overhead, regardless of whether the communication abstraction is shared address access or messages. Good architecture and clever programming can reduce the unnecessary communication to a minimum, but still each algorithm has some level of inherent communication. It is an open question whether designers will be able to achieve the low miss penalties required for efficient communication while attempting to maximize processor performance through deep hierarchies. There are some positive indication in this direction. Many

---

1. Often in observing the widening processor-memory speed gap, comparisons are made between processor rates with memory access times, by taking the reciprocal of the access time. Comparing between throughput and latency in this way makes the gap appear artificially wider.

scientific applications have relatively poor cache behavior because they sweep through large data sets. Beginning with the IBM Power2 architecture and continuing in the SGI Power Challenge and Sun Ultrasparc, attention has been paid to improving the out of cache memory bandwidth, at least for sequential access. These efforts proved very valuable for database applications. So, we may be able to look forward to node memory structures that can sustain high bandwidth, even in the evolutionary scenario.

There are strong indications that multithreading will be utilized in future processor generations to hide the latency of local memory access. By introducing logical, thread-level parallelism on a single processor, this direction further reduces the cost of the transition from one processor to multiple processors, thus making small scale SMPs even more attractive on a broad scale. It also establishes an architectural direction that may yield much greater latency tolerance in the long term.

Link and switch bandwidths are increasing, although this phenomenon does not have the smooth evolution of CMOS under improving lithography and fabrication techniques. Links tend to advance through discrete technological changes. For example, copper links have transitioned through a series of driver circuits: unterminated lines carrying a single bit at a time were replaced by terminated lines with multiple bits pipelined on the wire, and these may be replaced by active equalization techniques. At the same time, links have gotten wider as connector technology has improved, allowing finer pitched, better matched connections, and cable manufacturing has advanced, providing better control over signal skew. For several years, it has seemed that fiber would soon take over as the technology of choice for high speed links. However, the cost of transceivers and connectors has impeded its progress. This may change in the future, as the efficient LED arrays that have been available in Gallium Arsinide (GAs) technologies become effective in CMOS. The real driver of cost reducer, of course, is volume. The arrival of gigabit ethernet, which uses the FiberChannel physical link, may finally drive the volume of fiber transceivers up enough to cause a dramatic cost reduction. In addition, high quality parallel fiber has been demonstrated. Thus, flexible high performance links with small physical cross section may provide an excellent link technology for some time to come.

The bandwidths that are being required even for a uniprocessor design, and the number of simultaneous outstanding memory transactions needed to obtain this bandwidth, are stretching the limits of what can be achieved on a shared bus. Many system designs have already streamlined the bus design by requiring all components to transfer entire cache lines. Adapters for I/O devices are constructed with one block caches that support the cache coherency protocol. Thus, essentially all systems will be constructed as SMPs, even if only one processor is attached. Increasingly, the bus is being replaced by a switch and the snooping protocols are being replaced by directories. For example the HP/Convex Exemplar uses a cross-bar, rather than a bus, with the PA8000 processor, the Sun Ultrasparc UPA has a switched interconnect within the 2-4 processor node, although these are connected by a packet switched bus in the Enterprise 6000. The IBM PowerPC-based G30 uses a switch for the data path, but still uses a shared bus for address and snoop. Where busses are still used, they are packet-switched (split-phase). Thus, even in the evolutionary scenario, we can expect to see high performance network integrated ever more deeply into high volume designs. This trend makes the transition from high volume, moderate scale parallel systems to large scale, moderate volume parallel system more attractive, because the new technology required is less.

Higher speed networks are a dominant concern for current I/O subsystems, as well. There has been a great deal of attention paid to improved I/O support, with PCI replacing traditional vendor

I/O busses. There is a very strong desire to support faster local area networks, such as gigabit ethernet, OC12 ATM (622 Mb/s), SCI, FiberChannels and P1394.2. A standard PCI bus can provide roughly one Gb/s of bandwidth. Extended PCI with 64 bit, 66 MHz operation exists and promises to become more widespread in the future, offering multi-gigabit performance on commodity machines. Several vendors a looking at ways of providing direct memory bus access for high performance interconnects or distributed shared memory extensions.

These trends will ensure that small scale SMPs continue to be very attractive, and that clusters and more tightly packaged collections of commodity nodes will remain a viable option for the large scale. It is very likely that these designs will continue to improve as high-speed network interfaces become more mature. We already seeing a trend toward better integration of NICs with the cache coherence protocols so that control registers can be cached and DMA can be performed directly on user level data structures. For many reasons, large scale designs are likely to use SMPs nodes, so clusters of SMPs are likely to be a very important vehicle for parallel computing. With the recent introduction of the CC-NUMA based designs, such as the HP/Convex SPP, the SGI Origin, and especially the PentiumPro-based machines, large scale cache-coherent designs look increasingly attractive. The core question is whether a truly composable SMP-based node will emerge, so that large clusters of SMPs can essentially be snapped together as easily as one adds memory or I/O devices to a single node.

12.1.2 Hitting a Wall

So, if current trends hold the evolution of parallel computer architecture looks bright. Why might this not happen? Might we hit a wall instead? There are three basic possibilities, a latency wall, an overhead wall, and a cost or power wall.

The latency wall fundamentally is the speed of light, or rather of the propagation of electrical signals. We will soon see processors operating at clock rates in excess of 1 GHz, or a clock period of less that one nanosecond. Signals travel about a foot per nanosecond. In the evolutionary view, the physical size of the node does not get much smaller; it gets faster with more storage, but it is still several chips with connectors and PC board traces. Indeed, from 1987 to 97, the footprint of a basic processor and memory module did not shrink much. There was an improvement from two nodes per board to about four when first level caches moved on-chip and DRAM chips were turned on their side as SIMMs, but most designs maintained one level of cache off-chip. Even if the off-chip caches are eliminated (as in the Cray T3D and T3E), the processor chip consumes more an more power with each generation and a substantial amount of surface area is needed to dissipate the heat. Thus, thousand processor machines will still be meters across, not inches.

While the latency wall is real, there are several reasons why it will probably not impede the practical evolution of parallel architectures in the foreseeable future. One reason is that latency tolerance techniques at the processor level are quite effective on the scale of tens of cycles. There have been studies suggesting that caches are losing their effectiveness even on uniprocessors due to memory access latency[Bur*96]. However, these studies assume memory operations that are not pipelined and a processor typical of mid-90s designs. Other studies suggest that if memory operations are pipelined and if the processor is allowed to issue several instructions from a large instruction window, then branch prediction accuracy is a far more significant limit on performance than latency[JoPa97]. With perfect prediction, such an aggressive design can tolerate memory access times in the neighborhood of 100 cycles for many applications. Multithreading techniques provide an alternative source of instruction level parallelism that can be used to hide

latency, even with imperfect branch prediction. But, such latency tolerance techniques fundamentally demand bandwidth and bandwidth comes at a cost. The cost arises either through higher signalling rates, more wires, more pins, more real estate, or some combination of these. Also, the degree of pipelining that a component can support is limited by its occupancy. To hide latency requires careful attention to the occupancy of every stage along the path of access or communication. Where the occupancy cannot be reduced, interleaving techniques must be used to reduce the effective occupancy.

Following the evolutionary path, speed of light effects are likely to be dominated by bandwidth effects on latency. Currently, a single cache-line sized transfer is several hundred bits in length and, since links are relatively narrow, a single network transaction reaches entirely across the machine with fast cut-through routing. As links get wider, the effective length of a network transaction, i.e., the number of phits, will shrink, but there is quite a bit of room for growth before it takes more than a couple of concurrent transactions per processor to cover the physical latency. Moreover, cache block sizes are increasing just to amortize the cost of a DRAM access, so the length of a network transaction, and hence the number of outstanding transactions required to hide latency, may be nearly constant as machines evolve. Explicit message sizes are likely to follow a similar trend, since processors tend to be inefficient in manipulating objects smaller than a cache block.

Much of the communication latency today is in the network interface, in particular that store-and-forward delay at the source and at the destination, rather than in the network itself. The network interface latency is likely to be reduced as designs mature and as it becomes a larger fraction of the network latency. Consider, for example, what would be required to cut-through one or both of the network interfaces. On the source side, there is no difficulty in translating the destination to a route and spooling the message onto the wire as it becomes available from the processor. However, the processor may not be able to provide the data into the NI as fast as the NI spools data into the network, so as part of the link protocol it may be necessary to hold back the message (transferring idle phits on the wire). This machinery is already built into most switches. Machines such as the Intel Paragon, Meiko CS-2, and Cray T3D provide flow-control all the way through the NI and back to the memory system in order to perform large block transfers without a store-and-forward delay. Alternatively, it may be possible to design the communication assist such that once a small message starts onto the wire, e.g., a cache line, it is completely transferred without delay.

Avoiding the store-and-forward delay on the destination is a bit more challenging, because, in general, it is not possible to determine that the data in the message is good until the data has been received and checked. If it is spooled directly into memory, junk may be deposited. The key observation is that it is much more important that the address be correct than that the data contents be correct, because we do not want to spool data into the wrong place in memory. A separate checksum can be provided on the header. The header is checked before the message is spooled into the destination node. For a large transfer, there is typically a completion event, so that data can be spooled into memory and checked before being marked as arrived. Note that this does mean that the communication abstraction should not allow applications to poll data values within the bulk transfer to detect completion. For small transfers, a variety of tricks can be played to move the data into the cache speculatively. Basically, a line is allocated in the cache and the data is transferred, but if it does not checksum correctly, the valid bit on the line is never set. Thus, there will need to be greater attention paid to communication events in the design of the communication assist and memory system, but it is possible to streamline network transactions much more than the current state-of-the-art to reduce latency.

The primary reason that parallel computers will not hit a fundamental latency wall is that overall communication latency will continue to be dominated by overhead. The latency will be there, but it will still be a modest fraction of the actual communication time. The reason for this lies deep in the current industrial design process. Where there are one or more levels of cache on the processor chip, an off-chip cache, and then the memory system, in designing a cache controller for a given level of the memory hierarchy, the designer is given a problem which has a fast side toward the processor and a slow side toward the memory. The design goal is or minimize the expression:

$$\text{Ave Mem Access (S)} \ = \ \text{HitTime} \times \text{HitRate}_s + (1 - \text{HitRate}_s) \times \text{MissTime} \qquad \textbf{(EQ 12.1)}$$

for a typical address stream, S, delivered to the cache on the processor side.

This design goal presents an inherent trade-off because improvements in any one component generally come at the cost of worsening the others. Thus, along each direction in the design space the optimal design point is a compromise between extremes. The HitTime is generally fixed by the target rate of the fast component. This establishes a limit against which the rest of the design is optimized, i.e., the designer will do whatever is required to keep the HitTime within this limit. For example, one can consider cache organizational improvements to improve HitRate, such higher associativity, as long as it can be accomplished in the desired HitTime[Prz*88]. The critical aspect for parallel architecture concerns the MissTime. How hard is the designer likely to work to drive down the MissTime? The usual rule of thumb is to make the two additive components roughly equal. This guarantees that the design is within a factor of two of optimal and tends to be robust and good in practice. The key point is that since MissRates are small for a uniprocessor, the MissTime will be a large multiple of the HitTime. For first level caches with greater than 95% hit rates, it will be twenty times the HitTime and for lower caches it will still be an order of magnitude. A substantial fraction of the miss time is occupied by the transfer to the lower level of the storage hierarchy, and small additions to this have only a modest effect on uniprocessor performance. The cache designer will utilize this small degree of freedom in many useful ways. For example, cache line sizes can be increased to improve the HitRate, at the cost of a longer miss time.

In addition, each level of the storage hierarchy adds to the cost of the data transfer because another interface must be crossed. In order to modularize the design, interfaces tend to decouple the operations on either side. There is some cost to the handshake between caches on-chip. There is a larger cost in the interface between an on-chip cache and an off-chip cache, and a much larger cost to the more elaborate protocol required across the memory bus. In addition, for communication there is the protocol associated with the network itself. The accumulation of these effects is why the actual communication latency tends to be many times the speed of light bound. The natural response of the designer responsible for dealing with communication aspects of a design is invariably to increase the minimum data transfer size, e.g., increasing the cache line size or the smallest message fragment. This shifts the critical time from latency to occupancy. If each transfer is large enough to amortize the overhead, the additional speed of light latency is again a modest addition.

Wherever the design is partitioned into multiple levels of storage hierarchy with the emphasis placed on maximizing a level relative to the processor-side reference stream, the natural tendency of the designers will result in a multiplication of overhead with each level of the hierarchy between the processor and the communication assist. In order to get close to the speed of light latency limit, a very different design methodology will need to be established for processor

design, cache design, and memory design. One of the architectural trends that may bring about this change is the use of extensive out-of-order execution or multithreading to hide latency, even in uniprocessor systems. These techniques change the cache designer's goal. Instead of minimizing the sum of the two components in Equation 12.1, the goal is essentially to minimize the maximum, as in Equation 12.2.

$$max((\text{HitTime} \times \text{HitRate}_s), (1 - \text{HitRate}_s) \times \text{MissTime}) \qquad \textbf{(EQ 12.2)}$$

When there is a miss, the processor does not wait for it to be serviced, it continues executing and issues more requests, many of which, hopefully, hit. In the meantime, the cache is busy servicing the miss. Hopefully, the miss will be complete by the time another miss is generated or the processor runs out of things it can do without the miss completing. The miss needs to be detected and dispatched for service without many cycles of processor overhead, even though it will take some time to process it. In effect, the miss needs to be handed off for processing essentially within the HitTime budget.

Moreover, it may be necessary to sustain multiple outstanding requests to keep the processor busy, as fully explained in Chapter 11. The MissTime may be too large for a one-to-one balance in the components of Equation 12.2 to be met, either because of latency or occupancy effects. Also, misses and communication events tend to cluster, so the interval between operations that need servicing is frequently much less than the average.

"Little's Law" suggests that there is another potential wall, a cost wall. If the total latency that needs to be hidden is L and the rate of long latency requests is ρ, then the number of outstanding requests per processor when the latency is hidden is ρL, or greater when clustering is considered. With this number of communication events in-flight, the total bandwidth delivered by the network with P processors needs to be $P\rho L(P)$, where $L(P)$ reflects the increase in latency with machine size. This requirement establishes a lower bound on the cost of the network. To deliver this bandwidth, the aggregate bandwidth of the network itself will need to be much higher, as discussed in Chapter 10, since there will be bursts, collisions, and so on. Thus, to stay on the evolutionary path, latency tolerance will need to be considered in many aspects of the system design and network technology will need to improve in bandwidth and in cost.

12.1.3 Potential Breakthroughs

We have seen so far a rosy evolutionary path for the advancement of parallel architecture, with some dark clouds that might hinder this advance. Is there also a silver lining? Are there aspects of the technological trends that may create new possibilities for parallel computer design? The answer is certainly in the affirmative, but the specific directions are uncertain at best. While it is possible that dramatic technological changes, such as quantum devices, free space optical interconnects, molecular computing, or nanomechanical devices are around the corner, there appears to be substantial room left in the advance of conventional CMOS VLSI devices[Patt95]. The simple fact of continued increase in the level of integration is likely to bring about a revolution in parallel computer design.

From an academic viewpoint, it is easy to underestimate the importance of packaging thresholds in the process of continued integration, but, history shows that these factors are dramatic indeed. The general effect of the thresholds of integration is illustrated in Figure 12-2, that shows two qualitative trends. The straight line reflects the steady increase in the level of systems integration

with time. Overlaid on this is a curve depicting the amount of innovation in system design. A given design regime tends to be stable for a considerable period, in spite of technological advance, but when the level of integration crosses a critical threshold many new design options are enabled and there is a design renaissance. The figure shows two of the epochs in the history of computer architecture.

Recall, during the late 70's and early 80's computer system design followed a stable evolutionary path with clear segments: minicomputers, dominated by the DEC Vax in engineering and academic markets, mainframes, dominated by IBM in the commercial markets, vector supercomputers, dominated by Cray Research in the scientific market. The minicomputer had burst on the scene as a result of an earlier technology threshold, where MSI and LSI components, especially semiconductor memories, permitted the design of complex systems with relatively little engineering effort. In particular, this level of integration permitted the use of microprogramming techniques to support a large virtual address space and complex instruction set. The vector supercomputer niche reflected the end of a transition. Its exquisite ECL circuit design, coupled with semiconductor memory in a clean load-store register based architecture wiped out the earlier, more exotic parallel machine designs. These three major segments evolved in a predictable, evolutionary fashion, each with their market segment, while the microprocessor marched forward from 4-bits, to 8-bits, to 16-bits, bringing forward the personal computer and the graphics workstation.
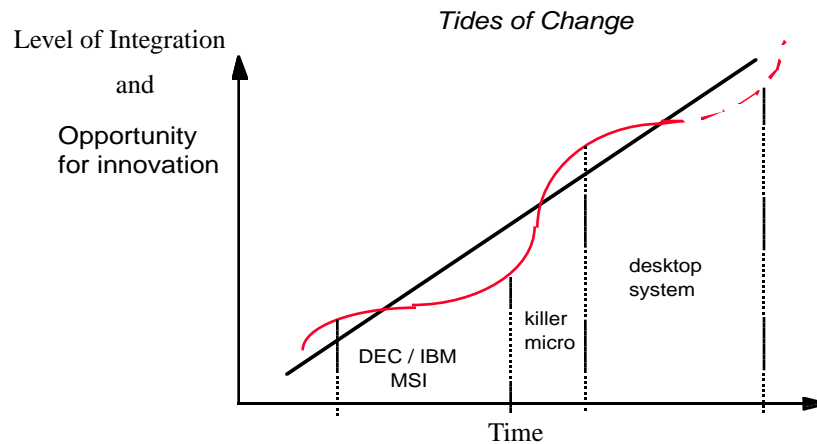

Figure 12-2 Tides of Change

The history on computing oscillates between periods of stable development and rapid innovation. Enabling technology generally does not hit "all of a sudden," it arrives and evolves. The "revolutions" occur when technology crosses a key threshold. One example is the arrival of the 32-bit microprocessor in the mid-80s which broke the stable hold of the less integrated minicomputers and mainframes, and enabled a renasaince in computer design, including low-level parallelism on-chip and high-level parallelism through multiprocessor designs. In the late 90's this transition is fully mature and microprocessor-based desktop and server technology dominate all segments of the market. There is tremendous convergence in parallel machine design. However, the level of integration continues to rise and soon the single-chip computer will be as natural as the single board computer of the 80's. The question is what new renasaince of design will this enable.

In the mid-80s, the microprocessor reached a critical threshold where a full 32-bit processor fit on a chip. Suddenly the entire picture changed. A complete computer of significant performance

and capacity fit on a board. Several such boards could be put in a system. Suddenly, bus-based cache-coherent SMPs appeared from man small companies, including Synapse, Encore, Flex, and Sequent. Relatively large message passing systems appeared from Intel, nCUBE, Ametek, Inmos, and Thinking Machines Corp. At the same time, several minisupercomputer vendors appeared, including Multiflow, FPS, Culler Scientific, Convex, Scientific Computing System, and Cydrome. Several of these companies failed as the new plateau was established. The workstation and later the personal computer absorbed the technical computing aspect of the minicomputer market. SMP servers took over the larger scale data centers, transaction processing, and engineering analysis, eliminating the minisuper. The vector supercomputers gave way to massively parallel microprocessor-based systems. Since that time, the evolution of designs has again stabilized. The understanding of cache coherence techniques has advanced, allowing shared address support at increasingly large scale. The transition of scalable, low latency networks from MPPs to conventional LAN or computer room environments has allowed casually engineered clusters of PCs, workstations, or SMPs to deliver substantial performance at very low cost, essentially as a personal supercomputer. Several very large machines are constructed as clusters of shared memory machines of various sizes. The convergence observed throughout this book is clearly in progress and the basic design question facing parallel machine designers is "how will the commodity components be integrated?" not what components will be used.

Meanwhile, *the level of integration in microprocessors and memories is fast approaching a new critical threshold where a complete computer fit on a chip, not a board.* Soon after the turn of the century the gigabit DRAM chip will arrive. Microprocessors are on the way to 100 Million transistors by the turn of the century. This new threshold is likely to bring about a new design renaissance as profound as that of the 32-bit microprocessor of the mid-80s, the semiconductor memory of the mid-70s, and the IC of the 60's. Basically, the strong differentiation between processor chips and memory chips will break down, and most chips will have processing logic and memory.

It is easy to enumerate many reasons why the processor-and-memory level of integration will take place and is likely to enable dramatic change in computer design, especially parallel computer design. Several research projects are investigating aspects of this new design space, under a variety of acronyms (PIM, IRAM, C-RAM, etc.). To avoid confusion, we will give processor-and-memory yet another, PAM. Only history will reveal which old architectural ideas will gain new life and which completely new ideas will arrive. Let's look at some of the technological factors leading toward new design options.

One clear factor is that microprocessors chips are mostly memory. It is SRAM memory used for caches, but memory none-the-less. Table 12-1 shows the fraction of the transistors and die area used for caches and memory interface, including store buffers, etc. for four recent microprocessors from two vendors.[Pat*97] The actual processor is a small and diminishing component of the microprocessor chip, even though processors are getting quite complicated. This trend is

made even more clear by Figure 12-3, which shows the fraction of the transistors devoted to caches in several microprocessors over the past decade[Burg97].

Table 12-1 Fraction of Microprocessor Chips devoted to Memory

Year	Micro-processor	On-Chip Cache Size	Total Transistors	fraction devoted to memory	Die Area (mm^2)	fraction devoted to memory
1993	Intel Pentium	I: 8 KB, D: 8 KB	3.1 M	32%	~300	32%
1995	Intel Pentium Pro	I: 8 KB, D: 8 KB, L2: 512 KB	P: 5.5 M +L2: 31 M	P: 18% +L2: 100% (Total: 88%)	P: 242 +L2: 282	P: 23% +L2: 100% (total: 64%)
1994	Digital Alpha 21164	I: 8 KB, D: 8 KB, L2: 96 KB	9.3 M	77%	298	37%
1996	Digital Strong-Arm SA-110	I: 16 KB D: 16 KB	2.1 M	95%	50	61%

The vast majority of the real estate, and an even larger fraction of the transistors, are used for data storage and organized as multiple level of on-chip caches. This investment in on-chip storage is necessary because of the time to access to off-chip memory, i.e., the latency of chip interface, off-chip caches, memory bus, memory controller, and the actual DRAM. For many applications, the best way to improve performance is to increase the amount of on-chip storage.

One clear opportunity this technological trend presents is putting multiple processors on-chip. Since the processor is only a small fraction of the chip real-estate, the potential peak performance can be increased dramatically at a small incremental cost. The argument for this approach is further strengthened by the diminishing returns in performance for processor complexity. For example, the real-estate devoted to register ports, instruction prefetch window, and hazard detection, and bypassing each increase more than linearly with the number of instructions issued per cycle, while the performance improves little beyond 4-way issue superscalar. Thus, for the same area, multiple processors of a less aggressive design can be employed[Olu*96]. This motivates re-examination of sharing issues that have evolved along with technology on SMPs since the mid-80s. Most of the early machines shared an off-chip first level cache, then there was room for separate caches, then L1 caches moved on-chip, and L2 caches were sometimes shared and sometimes not. Many of the basic trade-offs remain the same in the board-level and chip-level multiprocessors: sharing caches closer to the processor allows for finer grained data sharing and eliminates further levels of coherence support, but increases access time due to the interconnect on the fast side of the shared cache. Sharing at any level presents the possibility of positive or negative interference, depending on the application usage pattern. However, the board level designs were largely determined by the particular properties of the available components. With multiple processors on-chip, all the design options can be considered within the same homogeneous medium. Also, the specific trade-offs are different because of the different costs and per-
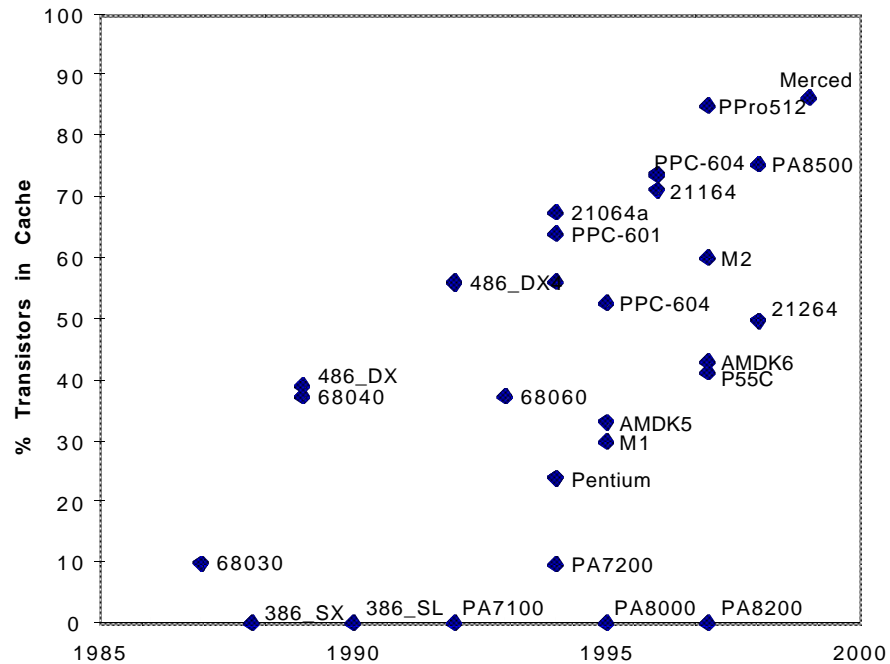
Figure 12-3 Fraction of Transistors on Microprocessors devote to Caches.

Since caches migrated on-chip in the mid 80's, the fraction of the transistors on commercial microprocessors that are devoted to caches has risen steadily. Although the processors are complex and exploit substantial instruction level parallelism, only be providing a great deal of local storage and exploiting locality can their bandwidth requirements be statisfied at reasonable latency.

formance characteristics. Given the broad emergence of inexpensive SMPs, especially with "glueless" cache coherence support, multiple processors on a chip is a natural path of evolution for multiprocessors.

A somewhat more radical change is suggested by the observation that the large volume of storage next to the processor could be DRAM storage, rather than SRAM. Traditionally, SRAM uses manufacturing techniques intended for processors and DRAM uses quite different techniques. The engineering requirements for microprocessors and DRAMs are traditionally very different. Microprocessor fabrication is intended to provide high clock rates and ample connectivity for datapaths and control using multiple layers of metal, whereas DRAM fabrication focuses on density and yield at minimal cost. The packages are very different. Microprocessors use expensive packages with many pins for bandwidth and materials designed to dissipate large amounts of heat. DRAM packages have few pins, low cost, and are suited to the low-power characteristics of DRAM circuits. However, these differences are diminishing. DRAM fabrication processes have become better suited for processor implementations, with two or three levels of metal, and better logic speed[Sau*96].

The drive toward integrating logic into the DRAM is driven partly by necessity and partly by opportunity. The immense increase in capacity (4x every three years) has required that the inter-

nal organization of the DRAM and its interface change. Early designs consisted of a single, square array of bits. The address was presented in two pieces, so a row could be read from the array and then a column selected. As the capacity increased, it was necessary to place several smaller arrays on the chip and to provide an interconnect from the many arrays and the pins. Also, with limited pins and a need to increase the bandwidth, there was a desire to run part of the DRAM chip at a higher rate. Many modern DRAM designs, including Synchronous DRAM, Enhanced DRAM, and RAMBUS make effective use of the row buffers within the DRAM chip and provide high bandwidth transfers between the row buffers and the pins. These approaches required that DRAM processes be capable of supporting logic as well. At the same time, there were many opportunities to incorporate new logic functions into the DRAM, especially for graphics support in video RAMs. For example, 3D-RAM places logic for z-buffer operations directly in the video-RAM chip that provides the frame buffer.

The attractiveness of integrating processor and memory is very much a threshold phenomenon. While processor design was constrained by chip area there was certainly no motivation to use fabrication techniques other than those specialized for fast processors, and while memory chips were small, so many were used in a system, there was no justification for the added cost of incorporating a processor. However, the capacity of DRAMs has been increasing more rapidly than the transistor count or, more importantly, the area used for processors. At the gigabit DRAM or perhaps the following generation, the incremental cost of the processor is modest, perhaps 20%. From the processor designer's viewpoint, the advantage of DRAM over SRAM is that it has more than an order of magnitude better density. However, the access time is greater, access is more restrictive, and refresh is required[Sau*96].

Somewhat more subtle threshold phenomena further increase the attractiveness of PAM. The capacity of DRAM has been growing faster than the demand for storage in most applications. The rapid increase in capacity has been beneficial at the high end, because it became possible to run very large problems, which tends to reduce the communication-to-computation ratio and make parallel processing more effective. At the low end it had the effect of reducing the number of memory chips sold per system. When there are only a few memory chips, traditional DRAM interfaces with few pins do not work well, so new DRAM interfaces with high speed logic are essential. When there is only one memory chip in a typical system, there is a huge cost savings in bring the processor on-chip and eliminating everything in between. However, this raises a question of what the memory organization should be for larger systems.

Augmenting the impact of critical thresholds of evolving technology are new technological factors and market changes. One is high speed CMOS serial links. ASIC cells are available that will drive in excess of 1 Gb/s on a serial link and substantially higher rates have been demonstrated in laboratory experiments. Previously, these rates were only available with expensive ECL circuits or GAs technology. High speed links using a few pins provides a cost effective means of integrating PAM chips into a large system, and cab form the basis for the parallel machine interconnection network. A second factor is the advancement and widespread use of FPGA and other configurable logic technology. This makes it possible to fabricate a single building block with processor, memory, and unconfigured logic, which can then be configured to suit a variety of applications. The final factor is the development of low power microprocessors for the rapidly growing market of network appliances, sometimes called WebPCs or Java stations, palmtop computers, and other sophisticated electronic devices. For many of these applications, modest single chip PAMs provide ample processing and storage capacity. The huge volume of these markets may indeed make PAM the commodity building block, rather than the desktop system.

The question presented by these technological opportunities is how the organization structure of the computer node should change. The basic starting point is indicated by Figure 12-4, which shows that each of the subsystems between the processor and the DRAM bit array present a narrow interface because pins and wires are expensive, even though they are relatively wide internally, and add latency. Within the DRAM chips, the datapath is extremely wide. The bit array itself is a collection of incredibly tightly packed trench capacitors, so little can be done there. However, the data buffers between the bit array and the external interface are still wide, less dense, and are essentially SRAM and logic. Recall, when a DRAM is read, a portion of the address is used to select a row, which is read into the data buffer, and then another portion of the address is used to select a few bits from the data buffer. The buffer is written back to the row, since the read is destructive. On a write, the row is read, a portion is modified in the buffer, and eventually it is written back.
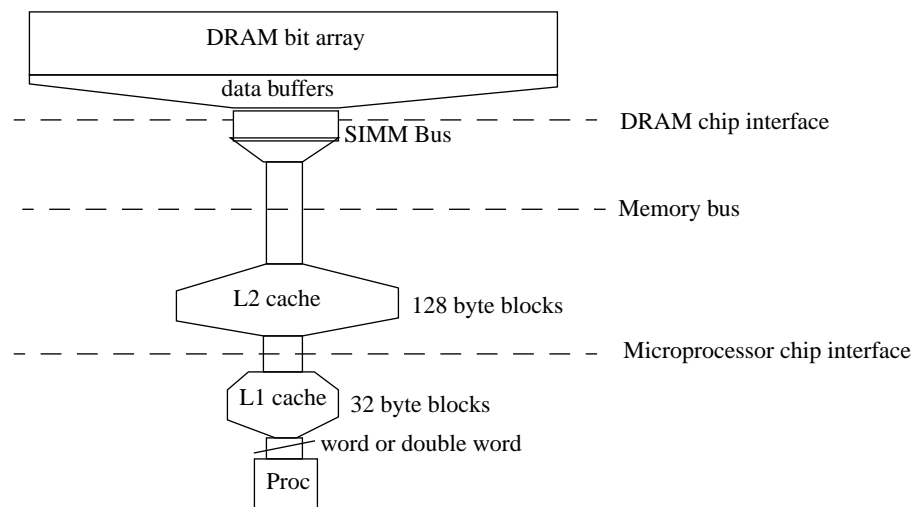

Figure 12-4 Bandwidths across a computer system

Processor data path is several words wide, but typically has a couple word wide interface to its L1 cache. The L1 cache blocks are 32 or 64 bytes wide, but it is constrained between the processor word-at-a-time operation and the microprocessor chip interface. The L2 cache blocks are even wider, but it is constrained between the microprocessor chip interface and the memory bus interface, both width critical. The SIMMS that form a bank of memory may have an interface that is wider and slower than the memory bus. Internally, this is a small section of a very wide data buffer, which is transferred directly to and from the actual bit arrays.

Current research investigates three basic possibilities, each of which have substantial history and can be understood, explored, and evaluated in terms of the fundamental design principles put forward in this book. They are surveyed briefly here, but the reader will surely want to consult the most recent literature and the web.

The first option is to place simple, dedicated processing elements into the logic associated with the data buffers of more-or-less conventional DRAM chips, indicated in Figure 12-5. This approach has been called Processor-in-Memory (PIM)[Gok*95] and Computational-RAM[Kog94,ElSn90]. It is fundamentally SIMD processing of a restricted class of data parallel operations. Typically, these will be small bit-serial processors providing basic logic operations, but they could operate on multiple bits or even a word at a time. As we saw in Chapter 1, the

approach has appeared several times in the history of parallel computer architecture. Usually it appears at the beginning of a technological transition when a general purpose operations are not quite feasible, so the specialize operations enjoys a generous performance advantage. Each time it has proved applicable for a limited class of operations, usually image processing, signal processing, or dense linear algebra, and each time it has given way to more general purpose solutions as the underlying technology evolves.

For example, in the early 60's, there were numerous SIMD machines proposed which would allow construction of a high performance machine by only replicating the function units and sharing a single instruction sequencer, including Staran, Pepe, and Illiac. The early effort culminated in the development of the Illiac IV at the University of Illinois which took many years and became operational only months before the Cray-1. In the early 80s the approach appeared again with the ICL DAP, which provided an array of compact processors, and got a big boost in the mid 80s when processor chips got large enough to support 32 bit-serial processors, but not a full 32-bit processor. The Goodyear MPP, Thinking Machines CM-1, and MasPar grew out of this window of opportunity. The key recognition that made the latter two machines much more successful that any previous SIMD approach was the need to provide a general purpose interconnect between the processing elements, not just a low dimensional grid, which is clearly cheap to build. Thinking Machines also was able to capture the arrival of the single chip floating point unit and modify the design in the CM-2 to provide operations on two thousand 32-bit PEs, rather than 64 thousand 1-bit PEs. However, these designs were fundamentally challenged by Amdahl's law, since the high performance mode can only be applied on the fraction of the problem that fits the specialized operations. Within a few years, they yielded to MPP designs with a few thousand general purpose microprocessors, which could perform SIMD operations and more general operation, so the parallelism could be utilized more of the time. The PIMs approach was deployed in the Cray 3/SSS, before the company filed "Chapter 11", to provide special support for the National Security Agency. It has also been demonstrated for more conventional technology[Aim96,Shi96].
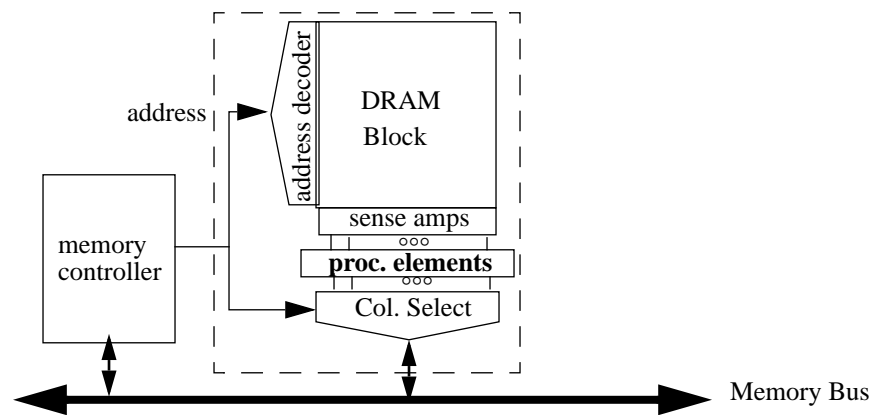

Figure 12-5 Processor-in-memory Organization

Simple function units (processing elements) are incorporated into the data buffers of fairly conventional DRAM chips.

A second option is to enhance the data buffers associated with banks of DRAM so they can be used as vector registers. There can be high bandwidth transfers between DRAM rows and vector

registers using the width of the bit arrays, but arithmetic is performed on vector registers by streaming the data through a small collection of conventional function units. This approach also rests on a good deal of history, including the very successful Cray machines, as well as several unsuccessful attempts. The Cray-1 success was startling in part because of the contrast with unsuccessful CDC Star-100 vector processor a year earlier. The Star-100 internally operated on short segments of data that were streamed out of memory into temporary registers. However, it provided vector operations only on contiguous (stride 1) vectors and the core memory it employed had long latency. The success of the Cray-1 was not just a matter of exposing the vector registers, but a combination of its use of fast semiconductor memory, providing a general interconnect between the vector registers and many banks of memory so that non-unit stride and later gather/scatter operations could be performed, and a very efficient coupling of the scalar and vector operations. In other words, low latency, high bandwidth for general access patterns, and efficient synchronization. Several later attempts at this style of design in newer technologies have failed to appreciate these lessons completely, including the FPS-DMAX, which provided linear combinations of vectors in memory, the Stellar, Ardent, and Stardent vector workstations, the Star-100 vector extension to the Sparcstation, the vector units in the memory controllers of the CM-5, and the vector function units on the Meiko CS-2. It is easy to become enamored with the peak performance and low cost of the special case, but if the start-up costs are large, addressing capability is limited, or interaction with scalar access is awkward, the fraction of time that the extension is actually used drops quickly and the approach is vulnerable to the general purpose solution.
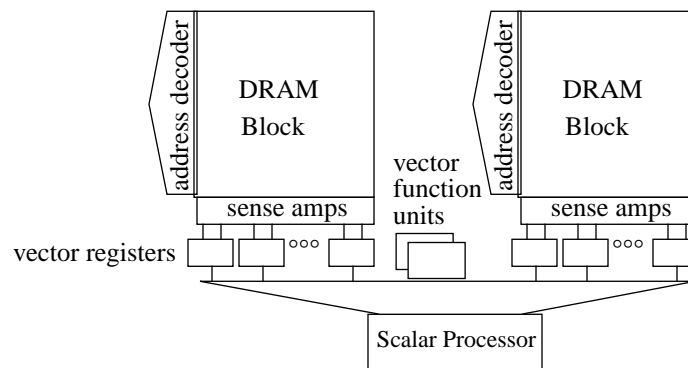

Figure 12-6 Vector IRAM Organization

DRAM data buffers are enhanced to provide vector registers, which can be streamed through pipelined function units. On-chip memory system and vector support is interfaced to a scalar processor, possibly with its own caches.

The third option pursues a general purpose design, but removes the layers of abstraction that have been associated with the distinct packaging of processors, off-chip caches, and memory systems. Basically, the data buffers associated with the DRAMs are utilized as the final layer of caches. This is quite attractive since advanced DRAM designs have essentially been using the data buffers as caches for several years. However, in the past they were restricted by the narrow interface out of the DRAM chips and the limited protocols of the memory bus. In the more integrated design, the DRAM-buffer caches can interface directly to the higher level caches of one or more on-chip processors. This approach changes the basic cache design trade-offs somewhat, but conventional analysis techniques apply. The cost of long lines is reduced, since the transfer between

the data buffer and the bit array is performed in parallel. The current high cost of logic near the DRAM tends to place a higher cost on associativity. Thus, many approaches use direct mapped DRAM-buffer caches and employ victim buffers or analogous techniques to reduce the rate of conflict misses.[Sau*96]. When these integrated designs are used as a building block for parallel machines, the expect effects are observed of long cache blocks causing increased false sharing along with improved data prefetching when spatial locality is present. [Nay*97].

When the PAM approach moves from the stage of academic, simulation based study to active commercial development we can expect to see an even more profound effect arising from the change in the design process. No longer is a cache designer in a position of optimizing one step in the path, constrained by a fast interface on one side and a slow interface on the other. The boundaries will have been removed and the design can be optimized as an end-to-end problem. All of the possibilities we have seen for integrating the communication assist are present: at the processor, into the cache controller, or into the memory controller, but in PAM then can be addressed in a uniform framework, rather than within the particular constraint of each component of the system.

Howsoever the detailed design of integrated processor and memory components shakes out, these new components are likely to provide a new, universal building block for larger scale parallel machines. Clearly, collections of them can be connected together to form distributed memory machines and the communication assist is likely to be much better integrated with the rest of the design, since it is all on one chip. Also, the interconnect pins are likely to be the only external interface, so communication efficiency will be even more critical. It will clearly be possible to build parallel machines on a scale far beyond what has never been possible. A practical limit which has stayed roughly constant since the earliest computers is that large scale systems are limited to about ten thousand components. Larger systems have been built, but tend to be difficult to maintain. In the early days, its was ten thousand vacuum tubes, then ten thousand gates, then ten thousand chips. Recently, large machines have had about a thousand processors and each processor required about ten components, either chips or memory SIMMS. The first of teraflops machine has almost ten thousand processors, each with multiple chips, so we will see if the pattern has changed. The complete system on a chip may well be the commodity building block of the future, used in all sorts of intelligent appliances at a volume much greater than the desktop, and hence at a much lower cost. In any case, we can look forward to much large scale parallelism as the processors per chip continues to rise.

A very interesting question is what happens when programs need access to more data than fits on the PAM. One approach is to provide a conventional memory interface for expansion. A more interesting alternative is to simply provide CC-NUMA access to other PAM chips, as developed in Chapter 8. The techniques are now very well understood, as is the minimal amount of hardware support required to provide this functionality. If the processor component of the PAM is indeed small, then even if only one process in the system is ever used, the additional cost of the other processors sitting near the memories is small. Since parallel software is become more and more widespread, the extra processing power is there when applications demand it.

12.2 Applications and System Software

The future is parallel computer architecture clearly is going to be increasingly a story of parallel software and of hardware/software interactions. We can view parallel software as falling into five

different classes: applications, compilers, languages, operating systems, and tools. In parallel software too, the same basic categories of change apply: evolution, hitting walls, and break-throughs.

12.2.1 Evolution

While data management and information processing are likely to be the dominant applications that exploit multiprocessors, applications in science and engineering have always been the proving ground for high end computing. Early parallel scientific computing focused largely on models of physical phenomena that result in fairly regular computational and communication characteristics. This allowed simple partitionings of the problems to be successful in harnessing the power of multiprocessors, just as it led earlier to effective exploitation of vector architectures. As the understanding of both the application domains and parallel computing grew, early adopters began to model the more complex, dynamic and adaptive aspects that are integral to most physical phenomena, leading to applications with irregular, unpredictable characteristics. This trend is expected to continue, bringing with it the attendant complexity for effective parallelization.

As multiprocessing becomes more and more widespread, the domains of its application will evolve, as will the applications whose characteristics are most relevant to computer manufacturers. Large optimization problems encountered in finance and logistics—for example, determining good crew schedules for commercial airlines—are very expensive to solve, have high payoff for corporations, and are amenable to scalable parallelism. Methods developed under the fabric of artificial intelligence, including searching techniques and expert systems, are finding practical use in several domains and can benefit greatly from increased computational power and storage. In the area of information management, an important trend is toward increased use of extracting trends and inferences from large volumes of data, using data mining and decision support techniques (in the latter, complex queries are made to determine trends that will provide the basis for important decisions). These applications are often computationally intensive as well as database-intensive, and mark an interesting marriage of computation and data storage/retrieval to build computation-and-information servers. Such problems are increasingly encountered in scientific research areas as well, for example in manipulating and analyzing the tremendous volumes of biological sequence information that is rapidly becoming available through the sciences of genomics and sequencing, and computing on the discovered data to produce estimates of three-dimensional structure. The rise of wide-scale distributed computing—enabled by the internet and the world-wide web—and the abundance of information in modern society make this marriage of computing and information management all the more inevitable as a direction of rapid growth. Multiprocessors have already become servers for internet search engines and world-wide web queries. The nature of the queries coming to an information server may also span a broader range, from a few, very complex mining and decision support queries at a time to a staggeringly large number of simultaneous queries in the case where the clients are home computers or hand-held information appliances.

As the communication characteristics of networks improve, and the need for parallelism with varying degrees of coupling becomes stronger, we are likely to see a coming together of parallel and distributed computing. Techniques from distributed computing will be applied to parallel computing to build a greater variety of information servers that can benefit from the better performance characteristics "within the box", and parallel computing techniques will be employed in the other direction to use distributed systems as platforms for solving a single problem in paral-

lel. Multiprocessors will continue to play the role of servers—database and transaction servers, compute servers, and storage servers—though the data types manipulated by these servers are becoming much richer and more varied. From information records containing text, we are moving to an era where images, three-dimensional models of physical objects, and segments of audio and video are increasingly stored, indexed, queried and served out as well. Matches in queries to these data types are often approximate, and serving them out often uses advanced compression techniques to conserve bandwidth, both of which require computation as well.

Finally, the increasing importance of graphics, media and real-time data from sensors—for military, civilian, and entertainment applications—will lead to increasing significance of real-time computing on data as it streams in and out of a multiprocessor. Instead of reading data from a storage medium, operating on it, and storing it back, this may require operating on data on its way through the machine between input and output data ports. How processors, memory and network integrate with one another, as well as the role of caches, may have to be revisited in this scenario. As the application space evolves, we will learn what kinds of applications can truly utilize large-scale parallel computing, and what scales of parallelism are most appropriate for others. We will also learn whether scaled up general-purpose architectures are appropriate for the highest end of computing, or whether the characteristics of these applications are so differentiated that they require substantially different resource requirements and integration to be cost-effective.

As parallel computing is embraced in more domains, we begin to see portable, pre-packaged parallel applications that can be used in multiple application domains. A good example of this is a an application called Dyna3D that solves systems of partial differential equations on highly irregular domains, using a method called the finite-element method [cite Hughes book, cite Dyna3D]. Dyna3D was initially developed at government research laboratories for use in modeling weapons systems on exotic, multi-million dollar supercomputers. After the cold war ended, the technology was transitioned into the commercial sector and it became the mainstay of crash modeling in the automotive industry and elsewhere on widely available parallel machines. By the time this book was written, the code was widely used on cost effective parallel servers for performing simulations on everyday appliances, such as what happens to a cellular phone when it is dropped.

A major enabling factor in the widespread use of parallel applications has been the availability of portable libraries that implement programming models on a wide range of machines. With the advent of the Message Passing Interface, this has become a reality for message passing, which is used in Dyna3D: The application of Dyna3D to different problems is usually done on very different platforms, from machines designed for message passing like the Intel Paragon to machines that support a shared address space in hardware like the Cray T3D to networks of workstations. Similar portability across a wide range of communication architecture performance characteristics has not yet been achieved for a shared address space programming model, but is likely soon.

As more applications are coded in both the shared address space and message passing models, we find a greater separation of the algorithmic aspects of creating a parallel program (decomposition and assignment) from the more mechanical and architecture-dependent aspects (orchestration and mapping), with the former being relatively independent of the programming model and architecture. Galaxy simulations and other hierarchical N-body computations, like Barnes-Hut, provide an example. The early message passing parallel programs used an orthogonal recursive bisection method to partition the computational domain across processors, and did not maintain a global tree data structure. Shared address space implementations on the other hand used a global tree, and a different partitioning method that led to a very different domain decomposition. With time, and with the improvement in message passing communication architectures, the message

passing versions also evolved to use similar partitioning techniques to the shared address space version, including building and maintaining a logically global tree using hashing techniques. Similar developments have occurred in ray tracing, where parallelizations that assumed no logical sharing of data structures gave way to logically shared data structures that were implemented in the message passing model using hashing. A continuation of this trend will also contribute to the portability of applications between systems that preferentially support one of the two models.

Like parallel applications, parallel programming languages are also evolving apace. The most popular languages for parallel computing continue to be based on the most popular sequential programming languages (C, C++, and FORTRAN), with extensions for parallelism. The nature of these extensions is now increasingly driven by the needs observed in real applications. In fact, it is not uncommon for languages to incorporate features that arise from experience with a particular class of applications, and then are found to generalize to some other classes of applications as well. In a similar vein, portable libraries of commonly occurring data structures and algorithms for parallel computing are being developed, and templates for these being defined for programmers to customize according to the needs of their specific applications.

Several styles of parallel languages have been developed. The least common denominator is MPI for message passing programming, which has been adopted across a wide range of platforms, and a sequential language enhanced with primitives like the ones we discussed in Chapter 2 for a shared address space. Many of the other language systems try to provide the programmer with a natural programming model, often based on a shared address space, and hide the properties of the machine's communication abstraction through software layers. One major direction has been explicitly parallel object-oriented languages, with emphasis on appropriate mechanisms to express concurrency and synchronization in a manner integrated with the underlying support for data abstraction. Another has been the development of data parallel languages with directives for partitioning such as High Performance Fortran, which are currently being extended to handle irregular applications. A third direction has been the development of implicitly parallel languages. Here, the programmer is not responsible for specifying the assignment, orchestration, or mapping, but only for the decomposition into tasks and for specifying the data that each task accesses. Based on these specifications, the runtime system of the language determines the dependences among tasks and assigns and orchestrates them appropriately for parallel execution on a platform. The burden on the programmer is decreased, or at least localized, but the burden on the system is increased.

With the increasing importance of data access for performance, several of these languages have proposed language mechanisms and runtime techniques to divide the burden of achieving data locality and reducing communication between the progammer and the system in a reasonable way. Finally, the increasing complexity of the applications being written and the phenomena being modeled by parallel applications has lead to the development of languages to support composability of bigger applications from smaller parts. We can expect much continued evolution in all these directions—appropriate abstractions for concurrency, data abstraction, synchronization and data management; libraries and templates, explicit and implicit parallel programming languages—with the goal of achieving a good balance between ease of programming, performance, and portability across a wide range of platforms (in both functionality and performance).

The development of compilers that can automatically parallelize programs has also been evolving over the last decade or two. With significant developments in the analysis of dependences among data accesses, compilers are now able to automatically parallelize simple array-based FORTRAN programs and achieve respectable performance on small-scale multiprocessors [cite].

Advances have also been made in compiler algorithms for managing data locality in caches and main memory, and in optimizing the orchestration of communication given the performance characteristics of an architecture (for example, making communication messages larger for message passing machines). However, compilers are still not able to parallelize more complex programs, especially those that make substantial use of pointers. Since the address stored in a pointer is not known at compile time, it is very difficult to determine whether a memory operation made through a pointer is to the same or a different address than another memory operation. In acknowledging this difficulty, one approach that is increasingly being taken is that of interactive parallelizing compilers. Here, the compiler discovers the parallelism that it can, and then gives intelligent feedback to the user about the places where it failed and asks the user questions about choices it might make in decomposition, assignment and orchestration. Given the directed questions and information, a user familiar with the program may have or may discover the higher-level knowledge of the application that the compiler did not have. Another approach in a similar vein is integrated compile-time and run-time parallelization tools, in which information gathered at runtime may be used to help the compiler—and perhaps user—parallelize the application successfully. or parallelization. Compiler technology will continue to evolve in these areas, as well as in the simpler area of providing support to deal with relaxed memory consistency models.

Operating systems are making the transition from treating uniprocessors to multiprocessors, and from treating multiprocessors as batch-oriented compute engines to multiprogrammed servers of computational and storage resources. In the former category, the evolution has included making operating systems more scalable by reducing the serialization within the operating system, and making the scheduling and resource management policies of operating systems take spatial and temporal locality into account (for example, not moving processes around too much in the machine, having them be scheduled close to the data they access, and having them be scheduled as far as possible on the same processor every time). In the latter category, a major challenge is managing resources and process scheduling in a way that strikes a good balance between fairness and performance, just as was done in uniprocessor operating systems. Attention is paid to data locality in scheduling application processes in a multiprogrammed workload, as well as to parallel performance in determining resource allocation: For example, the relative parallel performance of applications in a multiprogrammed workload may be used to determine how many processors and other resources to allocate to it at the cost of other programs. Operating systems for parallel machines are also increasingly incorporating the characteristics of multiprogrammed mainframe machines that were the mainstay servers of the past. One challenge in this area is exporting to the user the image of a single operating system (called single system image), while still providing the reliability and fault tolerance of a distributed system. Other challenges include containing faults so only the faulting application or the resources that the faulting application uses are affected by a fault, and providing the reliability and availability that people expect from mainframes. This evolution is necessary if scalable microprocessor-based multiprocessors are to truly replace mainframes as "enterprise" servers for large organizations, running multiple applications at a time.

With the increasing complexity of application-system interactions, and the increasing importance of the memory and communication systems for performance, it is very important to have good tools for diagnosing performance problems. This is particularly true of a shared address space programming model, since communication there is implicit and artifactual communication can often dominate other performance effects. Contention is a particularly prevalent and difficult performance effect for a programmer to diagnose, especially because the point in the program (or machine) where the effects of contention are felt can be quite different from the point that causes the contention. Performance tools continue to evolve, providing feedback about where in the pro-

gram are the largest overhead components of execution time, what data structures might be causing the data access overheads, etc. We are likely to see progress in techniques to increase the visibility of performance monitoring tools, which will be helped by the recent increasing willingness of machine designers to add a few registers or counters at key points in a machine solely for the purpose of performance monitoring. We are also likely to see progress in the quality of the feedback that is provided to the user, ranging from where in the program code a lot of time is being spent stalled on data access (available today), to which data structures are responsible for the majority of this time, to what is the cause of the problem (communication overhead, capacity misses, conflict misses with another data structure, or contention). The more detailed the information and the better it is cast in terms of the concepts the programmer deals with rather than in machine terms, the more likely that a programmer can respond to the information and improve performance. Evolution along this path will hopefully bring us to a good balance between software and hardware support for performance diagnosis.

A final aspect of the evolution will be the continued integration of the system software pieces described above. Languages will be designed to make the compiler's job easier in discovering and managing parallelism, and to allow more information to be conveyed to the operating system to make its scheduling and resource allocation decisions.

12.2.2 Hitting a Wall

On the software side, there is a wall that we have been hitting up against for many years now, which is the wall of programmability. While programming models are becoming more portable, architectures are converging, and good evolutionary progress is being made in many areas as described above, it is still the case that parallel programming is much more difficult than sequential programming. Programming for good performance takes a lot of work, sometimes in determining a good parallelization, and other times in implementing and orchestrating it. Even debugging parallel programs for correctness is an art and at best a primitive science. The parallel debugging task is difficult because of the interactions among multiple processes with their own program orders, and because of sensitivity to timing. Depending on when events in one process happen to occur relative to events in another process, a bug in the program may or may not manifest itself at runtime in a particular execution. And if it does, instrumenting the code to monitor certain events can cause the timing to be perturbed in such a way that the bug no longer appears. Parallel programming has been up against this wall for some time now; while evolutionary progress is helpful and has greatly increased the adoption of parallel computing, breaking down this wall will take a breakthrough that will truly allow parallel computing to realize the potential afforded to it by technology and architectural trends. It is unclear whether this breakthrough will be in languages per se, or in programming methodology, or whether it will simply be an evolutionary process.

12.2.3 Potential Breakthroughs

Other than breakthroughs in parallel programming languages or methodology and parallel debugging, we may hope for a breakthrough in performance models for reasoning about parallel programs. While many models have been quite successful at exposing the essential performance characteristics of a parallel system [Val90,Cul*96] and some have even provided a methodology for using these parameters, the more challenging aspect is modeling the properties of complex parallel applications and their interactions with the system parameters. There is not yet a well defined methodology for programmers or algorithm designers to use for this purpose, to deter-

mine how well an algorithm will perform in parallel on a system or which among competing partitioning or orchestration approaches will perform better. Another breakthrough may come from architecture, if we can somehow design machines in a cost-effective way that makes if much less important for a programmer to worry about data locality and communication; that is, to truly design a machine that can look to the programmer like a PRAM. An example of this would be if all latency incurred by the program could be tolerated by the architecture. However, this is likely to require tremendous bandwidth, which has a high cost, and it is not clear how to exploit the necessary concurrency from an application that is needed for this degree of latency tolerance, if the application has it to begin with.

The ultimate breakthrough, of course, will be the complete success of parallelizing compilers in taking a wide range of sequential programs and converting them into efficient parallel executions on a given platform, achieving good performance at a scale close to that inherently afforded by the application (i.e. close to the best one could do by hand). Besides the problems discussed above, parallelizing compilers tend to look for and utilize low-level, localized information in the program, and are not currently good at performing high-level, global analysis transformations. Compilers also lack semantic information about the application; for example, if a particular sequential algorithm for a phase of a problem does not parallelize well, there is nothing the compiler can do to choose another one. And for a compiler to take data locality and artifactual communication into consideration and manage the extended memory hierarchy of a multiprocessor is very difficult. However, even effective programmer-assisted compiler parallelization, keeping the programmer involvement to a minimum, would be perhaps the most significant software breakthrough in making parallel computing truly mainstream.

Whatever the future holds, it is certain that there will be continued evolutionary advance that will cross critical thresholds, there will be significant walls encountered, but there is likely to be ways around them, and there will be unexpected breakthroughs. Parallel computing will remain the place where exciting changes in computer technology and applications are first encountered, and there be an ever evolving cycle of hardware/software interactions.

12.3 References

[Bal*62]   J. R. Ball, R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, D. H. Shaffer, On the Use of the So-
            lomon Parallel-Processing Computer, Proceedings of the AFIPS Fall Joint Computer Conference,
            vol. 22, pp. 137-146, 1962.

[Bat79]   Kenneth E. Batcher, The STARAN Computer. Infotech State of the Art Report: Supercomputers.
            vol 2, ed C. R. Jesshope and R. W. Hockney, Maidenhead: Infotech Intl. Ltd, pp. 33-49.

[Bat80]   Kenneth E. Batcher, Design of a Massively Parallel Processor, IEEE Transactions on Computers,
            c-29(9):836-840.

[Bou*72]   W. J. Bouknight, S. A Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick,
            The Illiac IV System, Proceedings of the IEEE, 60(4):369-388, Apr. 1972.

[Bur*96]   D. Burger, J. Goodman, and A. Kagi, Memory Bandwidth Limitations in Future Microprocessors,
            Proc. of the 23rd Annual Symposium on Computer Architecture, pp. 78-89, May 1996.

[Burg97]   Doug Burger, System-Level Implications of Processor-Memory Integration, Workshop on Mix-
            ing Logic and DRAM: Chips that Compute and Remember at ISCA97, June 1997.

[Cor72]   J. A. Cornell. Parallel Processing of ballistic missile defense radar data with PEPE, COMPCON
            72, pp. 69-72.

[Cul*96]   D. E. Culler. R. M. Karp, R. M., D. A. Patterson, A., Sahay, K.E. Schauser, E. Santos, R. Subra-
            monian and T. von Eicken, LogP: A Practical Model of Parallel Computation, CACM. vol 39, no.
            11, pp. 78-85, Aug. 1996.

[Ell*92]   Duncan G. Elliott, W. Martin Snelgrove, and Michael Stumm. Computational RAM: A Memory-
            SIMD Hybrid and its Application to DSP. In Custom Integrated Circuits Conference, pages
            30.6.1--30.6.4, Boston, MA, May 1992.

[Ell*97]   Duncan Elliott, Michael Stumm , and Martin Snelgrove, Computational RAM: The case for SIMD
            computing in memory, Workshop on Mixing Logic and DRAM: Chips that Compute and Remem-
            ber at ISCA97, June 1997.

[Gok*95]   Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: the Terasys Massively Paral-
            lel PIM Array. Computer, 28(3):23--31, April 1995

[Hill85]   W. Daniel Hillis, The Connection Machine, MIT Press 1985.

[JoPa97]   Norman P. Jouppi and Parthasarathy Ranganathan, The Relative Importance of Memory Latency,
            Bandwidth, and Branch Limits to Performance, Workshop on Mixing Logic and DRAM: Chips
            that Compute and Remember at ISCA97, June 1997.

[Kog94]   Peter M. Kogge. EXECUBE - A New Architecture for Scalable MPPs. In 1994 International Con-
            ference on Parallel Processing, pages I77--I84, August 1994.

[Nic90]   Nickolls, J.R., The design of the MasPar MP-1: a cost effective massively parallel computer.
            COMPCON Spring '90 Digest of Papers. San Francisco, CA, USA, 26 Feb.-2 March 1990, p. 25-
            8.

[Olu*96]   Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson and Kunyung Chang, "The
            Case for a Single-Chip Multiprocessor", ASPLOS, Oct. 1996.

[Patt95]   Patterson, D., "Microprocessors in 2020," Scientific American, September 1995.

[Pat*97]   David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christo-
            foros Kozyrakis, Randi Thomas, and Katherine Yelick, A Case for Intelligent RAM , IEEE Micro,
            April 1997.

[Prz*88]   Steven Przybylski, Mark Horowitz, John Hennessy, Performance Tradeoffs in Cache Design,
            Proc. 15th Annual Symposium on Computer Architecture, May 1988, pp. 290-298.

[Red73]   S. F. Reddaway, DAP - a distributed array processor. First Annual International Symposium on Computer Architecture, 1973.

[Sau*96]   Ashley Saulsbury and Fong Pong and Andreas Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration", Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 90-101, May 1996.

[Slo*62]   Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds, The Solomon Computer, Proceedings of the AFIPS Fall Joint Computer Conference, vol. 22, pp. 97-107, 1962.

[Slot67]   Daniel L. Slotnick, Unconventional Systems, Proceedings of the AFIPS Spring Joint Computer Conference, vol 30, pp. 477-81, 1967.

[Sto70]   Harold S. Stone. A Logic-in-Memory Computer. IEEE Transactions on Computers, C-19(1):73--78, January 1970.

[TuRo88]   L.W. Tucker and G.G. Robertson, Architecture and Applications of the Connection Machine, IEEE Computer, Aug 1988, pp. 26-38

[Yam*94]   Nobuyuki Yamashita, Tohru Kimura, Yoshihiro Fujita, Yoshiharu Aimoto, Takashi Manaba, Shin'ichiro Okazaki, Kazuyuki Nakamura, and Masakazu Yamashina. A 3.84GIPS Integrated Memory Array Processor LSI with 64 Processing Elements and 2Mb SRAM. In International Solid-State Circuits Conference, pages 260--261, San Francisco, February 1994. NEC.

[Val90]   L. G., Valiant, A Bridging Model for Parallel Computation, CACM, vol 33, no. 8, pp. 103-11, Aug. 1990.

[ViCo78]   C.R. Vick and J. A. Cornell, PEPE Architecture - present and future. AFIPS Conference Proceedings, 47: 981-1002, 1978.

APPENDIX A   Parallel Benchmark Suites

Despite the difficulty of identifying "representative" parallel applications and the immaturity of parallel software (languages, programming environments), the evaluation of machines and architectural ideas must go on. To this end, several suites of parallel "benchmarks" have been developed and distributed. We put benchmarks in quotations because of the difficulty of relying on a single suite of programs to make definitive performance claims in parallel computing. Benchmark suites for multiprocessors vary in the kinds of application domains they cover, whether they include toy programs, kernels or real applications, the communication abstractions they are targeted for, and their philosophy toward benchmarking or architectural evaluation. Below we discuss some of the most widely used, publicly available benchmark/application suites for parallel processing. Table 0.1 on page 878 shows how these suites can currently be obtained.

A.1 ScaLapack

This suite [DW94] consists of parallel, message-passing implementations of the LAPACK linear algebra kernels. The Lapack suite consists of many important linear algebra kernels, including routines to solve linear systems of equations, eigenvalue problems, and singular value problems. matrix multiplications, factorizations and eigenvalue solvers. It therefore also includes many types of matrix factorizations (including LU and others) used by these routines. Information about the ScaLapack suite, and the suite itself, can be obtained from the Netlib repository maintained at the Oak Ridge National Laboratories.

A.2 TPC

The Transaction Processing Performance Council (TPC), founded in 1988, has created a set of publicly available benchmarks, called TPC-A, TPC-B and TPC-C, that are representative of different kinds of transaction-processing workloads and database system workloads <<cite>>. While database and transaction processing workloads are very important in actual usage of parallel machines, source codes for these workloads are almost impossible to obtain because of their competitive value to their developers.

TPC has a rigorous policy for reporting results. They use two metrics: (i) throughput in transactions per second, subject to a constraint that over 90% of the transactions have a response time less than a specified threshold; (ii) cost-performance in price per transaction per second, where price is the total system price and maintenance cost for five years. In addition to the innovation in metrics, they also provide benchmarks that scale with the power of the system in order to measure it realistically.

The first TPC benchmark was TPC-A, intended to consist of a single, simple, update-intensive transaction that provides a simple, repeatable unit of work, and is designed to exercise the key features of an on-line transaction processing (OLTP) system. The chosen transaction is from banking, and consists of reading 100 bytes from a terminal, updating the account, branch and teller records, and writing a history record, and finally writing 200 bytes to a terminal.

TPC-B, the second benchmark, is an industry-standard database (not OLTP) benchmark, designed to exercise the system components necessary for update-intensive database transactions. It therefore has significant disk I/O, moderate system and application execution time, and requires transaction integrity. Unlike OLTP, it does not require terminals or networking.

The third benchmark, TPC-C, was approved in 1992. It is designed to be more realistic than TPC-A but carry over many of its characteristics. As such, it is likely to displace TPC-A with time. TPC-C is a multi-user benchmark, and requires a remote terminal emulator to emulate a population of users with their terminals. It models the activity of a wholesale supplier with a number of geographically distributed sales districts and supply warehouses, including customers placing orders, making payments or making inquiries, as well as deliveries and inventory checks. It uses a database size that scales with the throughput of the system. Unlike TPC-A, which has a single, simple type of transaction that is repeated and therefore requires a simple database, TPC-C has a more complex database structure, multiple transaction types of varying complexity, online and deferred execution modes, higher levels of contention on data access and update, pat-

terns that simulate hot-spots, access by primary as well as non-primary keys, realistic requirements for full-screen terminal I/O and formatting, requirements for full transparency of data partitioning, and transaction rollbacks.

<<talk about TPC-D>>

TPC also plans to add some more benchmarks, including a benchmark representative of decision support systems (in which transactions manipulate large volumes of data in complex queries, and which provide timely answers to critical business questions), and one enterprise server benchmark (which provides concurrent OLTP and batch transactions, as well as heavyweight read-only OLTP transactions, with tighter response-time constraints). A client-server benchmark is also under consideration. Information about all these benchmarks can be obtained by contacting the Transaction Processing Performance Council.

A.3 SPLASH

The SPLASH (Stanford ParalleL Applications for SHared Memory) suite [SWG92] was developed at Stanford University to facilitate the evaluation of architectures that support a cache-coherent shared address space. It was replaced by the SPLASH-2 suite [SW+95], which enhanced some applications and added more. The SPLASH-2 suite currently contains 7 complete applications and 5 computational kernels. Some of the applications and kernels are provided in two versions, with different levels of optimization in the way data structures are designed and used (see the discussion of levels of optimization in Section 4.3.2). The programs represent various computational domains, mostly scientific and engineering applications and computer graphics. They are all written in C, and use the Parmacs macro package from Argonne National Laboratories [LO+87] for parallelism constructs. All of the parallel programs we use for workload-driven evaluation of shared address space machines in this book come from the SPLASH-2 suite.

A.4 NAS Parallel Benchmarks

The NAS benchmarks [BB+91] were developed by the Numerical Aerodynamic Simulation group at the National Aeronautic and Space Administration (NASA). They are a set of eight computations, five kernels and three pseudo-applications (not complete applications but more representative of the kinds of data structures, data movement and computation required in real aerophysics applications than the kernels). These computations are each intended to focus on some important aspect of the types of highly parallel computations in aerophysics applications. The kernels include an embarrassingly parallel computation, a multigrid equation solver, a conjugate gradient equation solver, a three-dimensional FFT equation solver, and an integer sort. Two different data sets, one small and one large, are provided for each benchmark. The benchmarks are intended to evaluate and compare real machines against one another. The performance of a machine on the benchmarks is normalized to that of a Cray Y-MP (for the smaller data sets) or a Cray C-90 (for the larger data sets).

The NAS benchmarks take a different approach to benchmarking than the other suites described in this section. Those suites provide programs that are already written in a high-level language

(such as Fortran or C, with constructs for parallelism). The NAS benchmarks, on the other hand, do not provide parallel implementations, but are so-called "paper-and-pencil" benchmarks. They specify the problem to be solved in complete detail (the equation system and constraints, for example) and the high-level method to be used (multigrid or conjugate gradient method, for example), but do not provide the parallel program to be used. Instead, they leave it up to the user to use the best parallel implementation for the machine at hand. The user is free to choose language constructs (though the language must be an extension of Fortran or C), data structures, communication abstractions and mechanisms, processor mapping, memory allocation and usage, and low-level optimizations (with some restrictions on the use of assembly language). The motivation for this approach to benchmarking is that since parallel architectures are so diverse and there is no established dominant programming language or communication abstraction that is most efficient on all architectures, a parallel implementation that is best suited to one machine may not be appropriate for another. If we want to compare two machines using a given computation or benchmark, we should use the most appropriate implementation for each machine. Thus, fixing the code to be used is inappropriate, and only the high-level algorithm and the inputs with which it should be run should be specified. This approach puts a greater burden on the user of the benchmarks, but is more appropriate for comparing widely disparate machines. Providing the codes themselves, on the other hand, makes the user's task easier, and may be a better approach for exploring architectural tradeoffs among a well-defined class of similar architectures.

Because the NAS benchmarks, particularly the kernels, are relatively easy to implement and represent an interesting range of computations for scientific computing, they have been widely embraced by multiprocessor vendors.

A.5 PARKBENCH

The PARKBENCH (PARallel Kernels and BENCHmarks) effort [PAR94] is a large-scale effort

Table 0.1 Obtaining Public Benchmark and Application Suites

Benchmark Suite	Communication Abstraction	How to Obtain Code or Information
ScaLapack	Message-passing	E-mail: netlib@ornl.gov
TPC	Either (not provided parallel)	Transaction Proc. Perf. Council
SPLASH	Shared Address Space (CC)	ftp/Web: mojave.stanford.edu
NAS	Either (paper-and-pencil)	Web: www.nas.nasa.gov
PARKBENCH	Message-Passing	E-mail: netlib@ornl.gov

to develop a suite of microbenchmarks, kernels and applications for benchmarking parallel machine, with at least an initial focus on explicit message-passing programs using Fortran77 with the Parallel Virtual Machine (PVM) [BD+91] library for communication. Versions of the programs in the High Performance Fortran language [HPF93] are also provided, to provide portability across message-passing and shared address space platforms.

PARKBENCH provides different the types of benchmarks we have discussed: low-level benchmarks or microbenchmarks, kernels, compact applications, and compiler benchmarks (the last two categories yet to be provided <<yes still?>>). The PARKBENCH committee proposes a methodology for choosing types of benchmarks to evaluate real machines that is similar to the one advocated here (except for the issue of using different levels of optimization). Among the

low-level benchmarks, the PARKBENCH committee first suggest using some existing benchmarks to measure per-node performance, and provide some of their own microbenchmarks for this purpose as well. The purpose of these uniprocessor microbenchmarks is to characterize the performance of various aspects of the architecture and compiler system, and obtain some parameters that can be used to understand the performance of the kernels and compact applications. The uniprocessor microbenchmarks include timer calls, arithmetic operations and memory bandwidth and latency stressing routines. There are also multiprocessor microbenchmarks that test communication latency and bandwidth—both point-to-point as well as all-to-all—as well as global barrier synchronization. Several of these multiprocessor microbenchmarks are taken from the earlier Genesis benchmark suite.

The kernel benchmarks are divided into matrix kernels (multiplication, factorization, transposition and tridiagonalization), Fourier transform kernels (a large 1-d FFT and a large 3-D FFT), partial differential equation kernels (a 3-d successive-over-relaxation iterative solver, and the multigrid kernel from the NAS suite), and others including the conjugate gradient, integer sort and embarrassingly parallel kernels from the NAS suite, and a paper-and-pencil I/O benchmark.

Compact applications (full but perhaps simplified applications) are intended in the areas of climate an meteorological modeling, computational fluid dynamics, financial modeling and portfolio optimization, molecular dynamics, plasma physics, quantum chemistry, quantum chromodynamics and reservoir modeling, among others. Finally, the compiler benchmarks are intended for people developing High Performance Fortran compilers to test their compiler optimizations, not to evaluate architectures.

A.6 Other Ongoing Efforts

SPEC/HPCG: The developers of the widely used SPEC (Standard Performance Evaluation Corporation) suite for uniprocessor benchmarking [SPE89] have teamed up with the developers of the Perfect Club (PERFormance Evaluation for Cost effective Transformations) Benchmarks for traditional vector supercomputers [BC+89] to form the SPEC: HPCG (High Performance Computing Group) to develop a suite of benchmarks measuring the performance of systems that "push the limits of computational technology", notably multiprocessor systems.

Many other benchmarking efforts exist. As we can see, most of them so far are for message-passing computing, though some of these are beginning to provide versions in High Performance Fortran that can be run on any of the major communication abstraction with the appropriate compiler support. We can expect the development of more shared address space benchmarks in the future. Many of the existing suites are also targeted toward scientific computing (the PARKBENCH and NAS efforts being the most large-scale among these), though there is increasing interest in producing benchmarks for other classes of workloads (including commercial and general purpose time-shared) for parallel machines. Developing benchmarks and workloads that are representative of real-world parallel computing and are also effectively portable to real machines is a very difficult problem, and the above are all steps in the right direction.