8 Multiprocessors

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. … Electronic circuits are ultimately limited in their speed of operation by the speed of light… and many of the circuits were already operating in the nanosecond range.

Bouknight et al., *The Illiac IV System* [1972]

… sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light . . .

A. L. DeCegama, *The Technology of Parallel Processing, Volume I* (1989)

… today's machines… are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.

Mitchell, *The Transputer: The Time Is Now* [1989]

8.1 Introduction

As the quotations that open this chapter show, the view that advances in uniprocessor architecture were nearing an end has been widely held at varying times. To counter this view, we observe that during the period 1985–95, uniprocessor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s. On balance, your authors believe that parallel machines will definitely have a bigger role in the future. This view is driven by three observations. First, since microprocessors are likely to remain the dominant uniprocessor technology, the logical way to improve performance beyond a single processor is by connecting multiple microprocessors together. This is likely to be more cost-effective than designing a custom processor. Second, it is unclear whether the pace of architectural innovation that has contributed to the rapid rate of performance growth starting in 1985 can be sustained indefinitely. As we saw in Chapter 4, modern multiple-issue processors have become incredibly complex, and the increases achieved in performance for increasing complexity and increasing silicon seem to be diminishing. Third, there appears to be slow but steady progress on the major obstacle to widespread use of parallel machines, namely software.

Your authors, however, are extremely reluctant to predict the death of advances in uniprocessor architecture. Indeed, we believe that the rapid rate of performance growth will continue at least into the next millennium. Whether this pace of innovation can be sustained longer is difficult to predict and hard to bet against. Nonetheless, if the pace of progress in uniprocessors does slow down, multiprocessor architectures will become increasingly attractive.

That said, we are left with two problems. First, multiprocessor architecture is a large and diverse field, and much of the field is in its infancy, with ideas coming and going and more architectures failing than succeeding. Given that we are already on page 636, full coverage of the multiprocessor design space and its tradeoffs would require another volume. Second, such coverage would necessarily entail discussing approaches that may not stand the test of time, something we have largely avoided to this point. For these reasons, we have chosen to focus on the mainstream of multiprocessor design: machines with small to medium numbers of processors (<100). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (>100 processors). The future architecture of such machines is so unsettled in the mid 1990s that even the viability of that marketplace is in doubt. In the past, the high-end scientific marketplace has been dominated by vector computers (see Appendix B), which in recent times have become small-scale parallel vector computers (typically 4 to 16 processors). There are several contending approaches and which, if any, will survive in the future remains unclear. We will return to this topic briefly at the end of the chapter, in section 8.10.

A Taxonomy of Parallel Architectures

We begin this chapter with a taxonomy so that you can appreciate both the breadth of design alternatives for multiprocessors and the context that has led to the development of the dominant form of multiprocessors. We briefly describe the alternatives and the rationale behind them; a longer description of how these different models were born (and often died) can be found in the historical perspectives at the end of the chapter.

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 30 years ago, Flynn proposed a simple model of categorizing all computers that is still useful today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the machine, and placed all computers in one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This is a uniprocessor.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. The processors are typically special purpose, since full generality is not required.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial machine of this type has been built to date, but may be in the future.
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors.

This is a coarse model, as some machines are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

As discussed in the historical perspectives, many of the early multiprocessors were SIMD, and the SIMD model received renewed attention in the 1980s. In the last few years, however, MIMD has clearly emerged as the architecture of choice for general-purpose multiprocessors. Two factors are primarily responsible for the rise of the MIMD machines:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user machines focusing on high performance for one application, as multiprogrammed machines running many tasks simultaneously, or as some combination of these functions.
2. MIMDs can build on the cost/performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and small, single-processor servers.

Existing MIMD machines fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. We refer to the machines by their memory organization, because what constitutes a small or large number of processors is likely to change over time.

The first group, which we call *centralized shared-memory architectures*, have at most a few dozen processors in the mid 1990s. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory can satisfy the memory demands of a small number of processors. Because there is a single main memory that has a uniform access time from each processor, these machines are sometimes called

UMAs for *uniform memory access*. This type of centralized shared-memory architecture is currently by far the most popular organization. Figure 8.1 shows what these machines look like. The architecture of such multiprocessors is the topic of section 8.3.

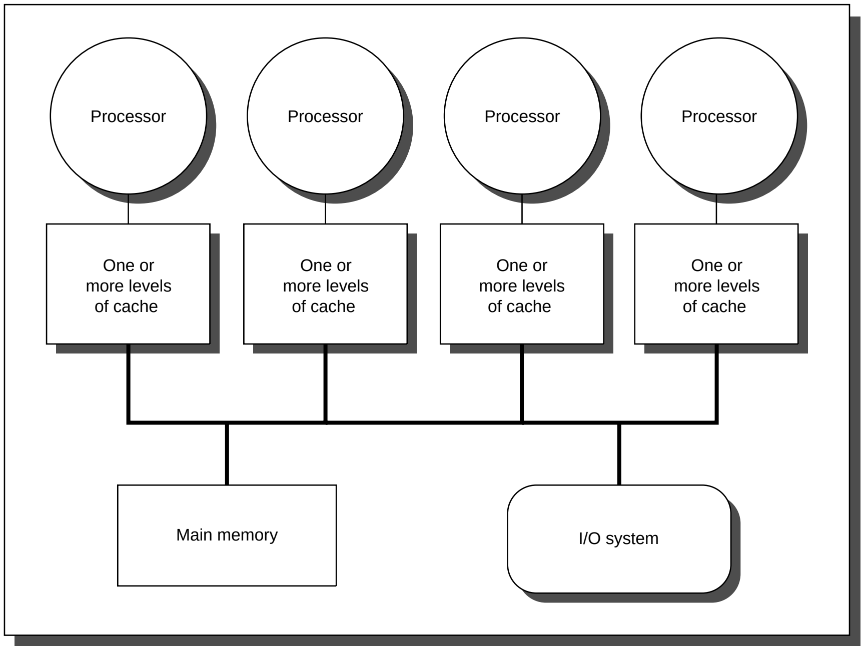

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

FIGURE 8.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus.

The second group consists of machines with physically distributed memory. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the scale of machine for which distributed memory is preferred over a single, centralized memory continues to decrease in number (which is another reason not to use small and large scale). Of course, moving to a distributed-memory organization raises the need for a high bandwidth interconnect, of which we saw examples in Chapter 7. Figure 8.2 shows what these machines look like.

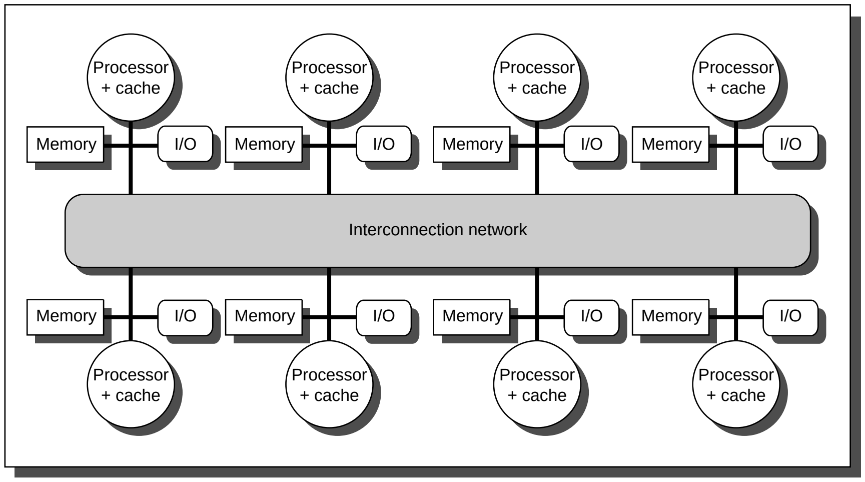


FIGURE 8.2 The basic architecture of a distributed-memory machine consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is often less scalable than the global interconnection network.

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth, if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantage for a distributed memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency because the processors no longer share a single centralized memory. As we will see shortly, the use of distributed memory leads to two different paradigms for interprocessor communication.

Typically, I/O as well as memory is distributed among the nodes of the multiprocessor, and the nodes may actually each contain a small number (2–8) of processors interconnected with a different technology. While this *clustering* of multiple processors in a node together with a memory and a network interface may be quite useful from a cost-efficiency viewpoint, it is not fundamental to how these machines work, and so we will focus on the one-processor-per-node style of machine throughout this chapter. The major architectural differences that distinguish among the distributed-memory machines are how communication is done and the logical architecture of the distributed memory.

Models for Communication and Memory Architecture

As discussed earlier, any large-scale multiprocessor must use multiple memories that are physically distributed with the processors. There are two alternative architectural approaches that differ in the method used for communicating data among processors. The physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These machines are called *distributed shared-memory* (*DSM*) or *scalable shared-memory* architectures. The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does *not* mean that there is a single, centralized memory. In contrast to the centralized memory machines, also known as UMAs (uniform memory access), the DSM machines are also called *NUMAs*, *non-uniform memory access*, since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor. In such machines, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory module is essentially a separate computer; therefore these machines have been called *multicomputers*. As pointed out in the concluding remarks of the previous chapter, these machines can even be completely separate computers connected on a local area network. For applications that require little or no communication and can make use of separate memories, such clusters of machines, whether in a closet or on desktops, can form a very cost-effective approach.

With each of these organizations for the address space, there is an associated communication mechanism. For a machine with a shared address space, that address space can be used to communicate data implicitly via load and store operations; hence the name *shared memory* for such machines. For a machine with multiple address spaces, communication of data is done by explicitly passing messages among the processors. Therefore, these machines are often called *message passing machines*.

In message passing machines, communication occurs by sending messages that request action or deliver data just as with the simple network protocols discussed in section 7.2. For example, if one processor wants to access or operate on data in a remote memory, it can send a message to request the data or to perform some operation on the data. In such cases, the message can be thought of as a *remote procedure call* (*RPC*). When the destination processor receives the message, either by polling for it or via an interrupt, it performs the operation or access on behalf of the remote processor and returns the result with a reply message. This type of message passing is also called *synchronous*, since the initiating processor sends a request and waits until the reply is returned before continuing. Software systems have been constructed to encapsulate the details of sending and receiving messages, including passing complex arguments or return values, presenting a clean RPC facility to the programmer.

Communication can also occur from the viewpoint of the writer of data rather than the reader, and this can be more efficient when the processor producing data knows which other processors will need the data. In such cases, the data can be sent directly to the consumer process without having to be requested first. It is often possible to perform such message sends asynchronously, allowing the sender process to continue immediately. Often the receiver will want to block if it tries to receive the message before it has arrived; in other cases, the reader may check whether a message is pending before actually trying to perform a blocking receive. Also the sender must be prepared to block if the receiver has not yet consumed an earlier message. The message passing facilities offered in different machines are fairly diverse. To ease program portability, standard message passing libraries (for example, message passing interface, or MPI) have been proposed. Such libraries sacrifice some performance to achieve a common interface.

Performance Metrics for Communication Mechanisms

Three performance metrics are critical in any communication mechanism:

1. *Communication bandwidth*—Ideally the communication bandwidth is limited by processor, memory, and interconnection bandwidths, rather than by some aspect of the communication mechanism. The bisection bandwidth is determined by the interconnection network. The bandwidth in or out of a single node, which is often as important as bisection bandwidth, is affected both by the architecture within the node and by the communication mechanism. How does the communication mechanism affect the communication bandwidth of a node? When communication occurs, resources within the nodes involved in the communication are tied up or occupied, preventing other outgoing or incoming communication. When this occupancy is incurred for each word of a message, it sets an absolute limit on the communication bandwidth. This limit is often lower than what the network or memory system can provide. Occupancy may also have a component that is incurred for each communication event, such as an incoming or outgoing request. In the latter case, the occupancy limits the communication rate, and the impact of the occupancy on overall communication bandwidth depends on the size of the messages.

2. *Communication latency*—Ideally the latency is as low as possible. As we saw in Chapter 7, communication latency is equal to

   Sender overhead + Time of flight + Transmission time + Receiver overhead

   Time of flight is preset and transmission time is determined by the interconnection network. The software and hardware overheads in sending and receiving messages are largely determined by the communication mechanism and its implementation. Why is latency crucial? Latency affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait. Overhead and occupancy are closely related, since many forms of overhead also tie up some part of the node, incurring an occupancy cost, which in turn limits bandwidth. Key features of a communication mechanism may directly affect overhead and occupancy. For example, how is the destination address for a remote communication named, and how is protection implemented? When naming and protection mechanisms are provided by the processor, as in a shared address space, the additional overhead is small. Alternatively, if these mechanisms must be provided by the operating system for each communication, this increases the overhead and occupancy costs of communication, which in turn reduce bandwidth and increase latency.

3. *Communication latency hiding*—How well can the mechanism hide latency by overlapping communication with computation or with other communication? Although measuring this is not as simple as measuring the first two, it is an important characteristic that can be quantified by measuring the running time on machines with the same communication latency but different support for latency hiding. We will see examples of latency hiding techniques for shared memory in sections 8.6 and 8.7. While hiding latency is certainly a good idea, it poses an additional burden on the software system and ultimately on the programmer. Furthermore, the amount of latency that can be hidden is application dependent. Thus it is usually best to reduce latency wherever possible.

Each of these performance measures is affected by the characteristics of the communications needed in the application. The size of the data items being communicated is the most obvious, since it affects both latency and bandwidth in a direct way, as well as affecting the efficacy of different latency hiding approaches. Similarly, the regularity in the communication patterns affects the cost of naming and protection, and hence the communication overhead. In general, mechanisms that perform well with smaller as well as larger data communication requests, and irregular as well as regular communication patterns, are more flexible and efficient for a wider class of applications. Of course, in considering any communication mechanism, designers must consider cost as well as performance.

Advantages of Different Communication Mechanisms

Each of these communication mechanisms has its advantages. For shared-memory communication, advantages include

- Compatibility with the well-understood mechanisms in use in centralized multiprocessors, which all use shared-memory communication.

- Ease of programming when the communication patterns among processors are complex or vary dynamically during execution. Similar advantages simplify compiler design.
- Lower overhead for communication and better use of bandwidth when communicating small items. This arises from the implicit nature of communication and the use of memory mapping to implement protection in hardware, rather than through the operating system.
- The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private. As we will see, caching reduces both latency and contention for accessing shared data.

The major advantages for message-passing communication include

- The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.
- Communication is explicit, forcing programmers and compilers to pay attention to communication. This process may be painful for programmers and compiler writers, but it simplifies the abstraction of what is costly and what is not and focuses attention on costly communication. (If you think this advantage is a mixed bag, that's OK; so do many others.)

Of course, the desired communication model can be created on top of a hardware model that supports either of these mechanisms. Supporting message passing on top of shared memory is considerably easier: Because messages essentially send data from one memory to another, sending a message can be implemented by doing a copy from one portion of the address space to another. The major difficulties arise from dealing with messages that may be misaligned and of arbitrary length in a memory system that is normally oriented toward transferring aligned blocks of data organized as cache blocks. These difficulties can be overcome either with small performance penalties in software or with essentially no penalties, using a small amount of hardware support.

Supporting shared memory efficiently on top of hardware for message passing is much more difficult. Without explicit hardware support for shared memory, all shared-memory references need to involve the operating system to provide address translation and memory protection, as well as to translate memory references into message sends and receives. Loads and stores usually move small amounts of data, so the high overhead of handling these communications in software severely limits the range of applications for which the performance of software-based shared memory is acceptable. An ongoing area of research is the exploration of when a software-based model is acceptable and whether a software-based mechanism is usable for the highest level of communication in a hierarchically structured system. One promising direction is the use of virtual memory mechanisms to share objects at the page level, a technique called *shared virtual memory;* we discuss this approach in section 8.7.

In distributed-memory machines, the memory model and communication mechanisms distinguish the machines. Originally, distributed-memory machines were built with message passing, since it was clearly simpler and many designers and researchers did not believe that a shared address space could be built with distributed memory. More recently, shared-memory communication has been supported in virtually every machine designed for the latter half of the 1990s. What hardware communication mechanisms will be supported in the very largest machines (called *massively parallel processors,* or *MPPs*), which typically have more than 100 processors, is unclear; shared memory, message passing, and hybrid approaches are all contenders. Despite the symbolic importance of the MPPs, such machines are a small portion of the market and have little or no influence on the mainstream machines with tens of processors. We will return to a discussion of the possibilities and trends for MPPs in the concluding remarks and historical perspectives at the end of this chapter.

Although centralized memory machines using a bus interconnect still dominate in terms of market size, long-term technical trends favor distributing memory even in moderate-scale machines; we'll discuss these issues in more detail at the end of this chapter. These distributed shared-memory machines are a natural extension of the centralized multiprocessors that dominate the market, so we discuss these architectures in section 8.4. One important question that we address there is the question of caching and coherence.

Challenges of Parallel Processing

Two important hurdles, both explainable with Amdahl's Law, make parallel processing challenging. The first has to do with the limited parallelism available in programs and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in parallel machines, as our first Example shows.

EXAMPLE Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

ANSWER Amdahl's Law is

$$\text{Speedup} = \frac{1}{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, while the fraction of enhanced mode is the time spent in parallel mode. Substituting into the equation above:

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus to achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential. Of course, to achieve linear speedup (speedup of *n* with *n* processors), the entire program must usually be parallel with no serial portions. (One exception to this is *superlinear speedup* that occurs due to the increased memory and cache available when the processor count is increased. This effect is usually not very large.) In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors. Exercise 8.2 asks you to extend Amdahl's Law to deal with such a case. ■

The second major challenge involves the large latency of remote access in a parallel machine. In existing machines, communication of data between processors may cost anywhere from 50 clock cycles to over 10,000 clock cycles, depending on the communication mechanism, the type of interconnection network, and the scale of the machine. Figure 8.3 shows the typical round-trip delays to retrieve a word from a remote memory for several different parallel machines.

The effect of long communication delays is clearly substantial. Let's consider a simple Example.

Machine	Communication mechanism	Interconnection network	Processor count	Typical remote memory access time
SPARCCenter	Shared memory	Bus	≤ 20	1 µs
SGI Challenge	Shared memory	Bus	≤ 36	1 µs
Cray T3D	Shared memory	3D torus	32–2048	1 µs
Convex Exemplar	Shared memory	Crossbar + ring	8–64	2 µs
KSR-1	Shared memory	Hierarchical ring	32–256	2–6 µs
CM-5	Message passing	Fat tree	32–1024	10 µs
Intel Paragon	Message passing	2D mesh	32–2048	10–30 µs
IBM SP-2	Message passing	Multistage switch	2–512	30–100 µs

FIGURE 8.3 Typical remote access times to retrieve a word from a remote memory. In the case of shared-memory machines, this is the remote load time. For a message-passing machine, the value shown is the time to send a message and reply to the message.

EXAMPLE Suppose we have an application running on a 32-processor machine, which has a 2000-ns time to handle reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which may be only slightly pessimistic. Processors are stalled on a remote request, and the cycle time of the processors is 10 ns. If the base CPI (assuming that all references hit in the cache) is 1.0, how much faster is the machine if there is no communication versus if 0.5% of the instructions involve a remote communication reference?

ANSWER The effective CPI for the machine with 0.5% remote references is

$$\begin{aligned}\text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost} \\ &= 1.0 + 0.5\% \times \text{Remote request cost}\end{aligned}$$

The Remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{2000\text{ ns}}{10\text{ ns}} = 200\text{ cycles}$$

Hence we can compute the CPI:

$$\text{CPI} = 1.0 + 0.5\% \times 200 = 2.0$$

The machine with all local references is 2.0/1.0 = 2 times faster. In practice, the performance analysis is much more complex, since some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be quite a bit worse, since contention caused by many references trying to use the global interconnect can lead to increased delays. ■

These problems—insufficient parallelism and long latency remote communication—are the two biggest challenges in using multiprocessors. The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance. Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer. For example, we can reduce the frequency of remote accesses with either hardware mechanisms, such as caching shared data, or with software mechanisms, such as restructuring the data to make more accesses local. We can try to tolerate the latency by using prefetch, which we examined in Chapter 5.

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, sections 8.3 and 8.4 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 8.5 discusses synchronization, which, because it inherently involves interprocessor communication, is an additional potential bottleneck. Section 8.6 talks about latency hiding techniques and memory consistency models for shared memory. Before we wade into these topics, it is helpful to have some understanding of the characteristics of parallel applications, both for better comprehension of the results we show using some of these applications and to gain a better understanding of the challenges in writing efficient parallel programs.

8.2 Characteristics of Application Domains

In earlier chapters, we examined the performance and characteristics of applications with only a small amount of insight into the structure of the applications. For understanding the key elements of uniprocessor performance, such as caches and pipelining, general knowledge of an application is often adequate. In parallel processing, however, the additional performance-critical characteristics—such as load balance, synchronization, and sensitivity to memory latency—often depend on high-level characteristics of the application. These characteristics include factors like how data is distributed, the structure of a parallel algorithm, and the spatial and temporal access patterns to data. Therefore at this point we take the time to examine the two different classes of workloads that we will use for performance analysis in this chapter.

This section briefly describes the characteristics of two different domains of multiprocessor workloads: individual parallel programs and multiprogrammed workloads with operating systems included. Other major classes of workloads are databases, fileservers, and transaction processing systems. Constructing realistic versions of such workloads and accurately measuring them on multiprocessors, including any OS activity, is an extremely complex and demanding process, at the edge of what we can do with performance modeling tools. Future editions of this book may contain characterizations of such workloads. Happily, there is some evidence that the parallel processing and memory system behaviors of database and transaction processing workloads are similar to those of large multiprogrammed workloads, which include the OS activity. For the present, we have to be content with examining such a multiprogramming workload.

Parallel Applications

Our parallel applications workload consists of two applications and two computational kernels. The kernels are an FFT (fast Fourier transformation) and an LU decomposition, which were chosen because they represent commonly used techniques in a wide variety of applications and have performance characteristics typical of many parallel scientific applications. In addition, the kernels have small code segments whose behavior we can understand and directly track to specific architectural characteristics.

The two applications that we use in this chapter are Barnes and Ocean, which represent two important but very different types of parallel computation. We briefly describe each of these applications and kernels and characterize their basic behavior in terms of parallelism and communication. We describe how the problem is decomposed for a distributed shared-memory machine; certain data decompositions that we describe are not necessary on machines that have a single centralized memory.

The FFT Kernel

The *fast Fourier transform* (FFT) is the key kernel in applications that use spectral methods, which arise in fields ranging from signal processing to fluid flow to climate modeling. The FFT application we study here is a one-dimensional version of a parallel algorithm for a complex-number FFT. It has a sequential execution time for n data points of $n \log n$. The algorithm uses a high radix (equal to $\sqrt{n}$) that minimizes communication. The measurements shown in this chapter are collected for a million-point input data set.

There are three primary data structures: the input and output arrays of the data being transformed and the roots of unity matrix, which is precomputed and only read during the execution. All arrays are organized as square matrices. The six steps in the algorithm are as follows:

1. Transpose data matrix.
2. Perform 1D FFT on each row of data matrix.

3. Multiply the roots of unity matrix by the data matrix and write the result in the data matrix.
4. Transpose data matrix.
5. Perform 1D FFT on each row of data matrix.
6. Transpose data matrix.

The data matrices and the roots of unity matrix are partitioned among processors in contiguous chunks of rows, so that each processor's partition falls in its own local memory. The first row of the roots of unity matrix is accessed heavily by all processors and is often replicated, as we do, during the first step of the algorithm just shown.

The only communication is in the transpose phases, which require all-to-all communication of large amounts of data. Contiguous subcolumns in the rows assigned to a processor are grouped into blocks, which are transposed and placed into the proper location of the destination matrix. Every processor transposes one block locally and sends one block to each of the other processors in the system. Although there is no reuse of individual words in the transpose, with long cache blocks it makes sense to block the transpose to take advantage of the spatial locality afforded by long blocks in the source matrix.

The LU Kernel

LU is an LU factorization of a dense matrix and is representative of many dense linear algebra computations, such as QR factorization, Cholesky factorization, and eigenvalue methods. For a matrix of size $n \times n$ the running time is n^3 and the parallelism is proportional to n^2. Dense LU factorization can be performed efficiently by blocking the algorithm, using the techniques in Chapter 5, which leads to highly efficient cache behavior and low communication. After blocking the algorithm, the dominant computation is a dense matrix multiply that occurs in the innermost loop. The block size is chosen to be small enough to keep the cache miss rate low, and large enough to reduce the time spent in the less parallel parts of the computation. Relatively small block sizes (8×8 or 16×16) tend to satisfy both criteria. Two details are important for reducing interprocessor communication. First, the blocks of the matrix are assigned to processors using a 2D tiling: the $\frac{n}{B} \times \frac{n}{B}$ (where each block is $B \times B$) matrix of blocks is allocated by laying a grid of size $p \times p$ over the matrix of blocks in a cookie-cutter fashion until all the blocks are allocated to a processor. Second, the dense matrix multiplication is performed by the processor that owns the *destination* block. With this blocking and allocation scheme, communication during the reduction is both regular and predictable. For the measurements in this chapter, the input is a 512×512 matrix and a block of 16×16 is used.

A natural way to code the blocked LU factorization of a 2D matrix in a shared address space is to use a 2D array to represent the matrix. Because blocks are allocated in a tiled decomposition, and a block is not contiguous in the address space in a 2D array, it is very difficult to allocate blocks in the local memories of the processors that own them. The solution is to ensure that blocks assigned to a processor are allocated locally and contiguously by using a 4D array (with the first two dimensions specifying the block number in the 2D grid of blocks, and the next two specifying the element in the block).

The Barnes Application

Barnes is an implementation of the Barnes-Hut n-body algorithm solving a problem in galaxy evolution. *N-body algorithms* simulate the interaction among a large number of bodies that have forces interacting among them. In this instance the bodies represent collections of stars and the force is gravity. To reduce the computational time required to model completely all the individual interactions among the bodies, which grow as n^2, n-body algorithms take advantage of the fact that the forces drop off with distance. (Gravity, for example, drops off as $1/d^2$, where d is the distance between the two bodies.) The Barnes-Hut algorithm takes advantage of this property by treating a collection of bodies that are "far away" from another body as a single point at the center of mass of the collection and with mass equal to the collection. If the body is far enough from any body in the collection, then the error introduced will be negligible. The collections are structured in a hierarchical fashion, which can be represented in a tree. This algorithm yields an $n \log n$ running time with parallelism proportional to n.

The Barnes-Hut algorithm uses an octree (each node has up to eight children) to represent the eight cubes in a portion of space. Each node then represents the collection of bodies in the subtree rooted at that node, which we call a *cell*. Because the density of space varies and the leaves represent individual bodies, the depth of the tree varies. 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. For every node in the tree it visits, the algorithm determines if the center of mass of the cell represented by the subtree rooted at the node is "far enough away" from the body. If so, the entire subtree under that node is approximated by a single point at the center of mass of the cell, and the force this center of mass exerts on the body is computed. On the other hand, if the center of mass is not far enough away, the cell must be "opened" and each of its subtrees visited. The distance between the body and the cell, together with the error tolerances, determines which cells must be opened. This force calculation phase dominates the execution time. This chapter takes measurements using 16K bodies; the criterion for determining whether a cell needs to be opened is set to the middle of the range typically used in practice.

Obtaining effective parallel performance on Barnes-Hut is challenging because the distribution of bodies is nonuniform and changes over time, making partitioning the work among the processors and maintenance of good locality of reference difficult. We are helped by two properties: the system evolves slowly; and because gravitational forces fall off quickly, with high probability, each cell requires touching a small number of other cells, most of which were used on the last time step. The tree can be partitioned by allocating each processor a subtree. Many of the accesses needed to compute the force on a body in the subtree will be to other bodies in the subtree. Since the amount of work associated with a subtree varies (cells in dense portions of space will need to access more cells), the size of the subtree allocated to a processor is based on some measure of the work it has to do (e.g., how many other cells does it need to visit), rather than just on the number of nodes in the subtree. By partitioning the octree representation, we can obtain good load balance and good locality of reference, while keeping the partitioning cost low. Although this partitioning scheme results in good locality of reference, the resulting data references tend to be for small amounts of data and are unstructured. Thus this scheme requires an efficient implementation of shared-memory communication.

The Ocean Application

Ocean simulates the influence of eddy and boundary currents on large-scale flow in the ocean. It uses a restricted red-black Gauss-Seidel multigrid technique to solve a set of elliptical partial differential equations. *Red-black Gauss-Seidel* is an iteration technique that colors the points in the grid so as to consistently update each point based on previous values of the adjacent neighbors. *Multigrid methods* solve finite difference equations by iteration using hierarchical grids. Each grid in the hierarchy has fewer points than the grid below, and is an approximation to the lower grid. A finer grid increases accuracy and thus the rate of convergence, while requiring more execution time, since it has more data points. Whether to move up or down in the hierarchy of grids used for the next iteration is determined by the rate of change of the data values. The estimate of the error at every time-step is used to decide whether to stay at the same grid, move to a coarser grid, or move to a finer grid. When the iteration converges at the finest level, a solution has been reached. Each iteration has n^2 work for an $n \times n$ grid and the same amount of parallelism.

The arrays representing each grid are dynamically allocated and sized to the particular problem. The entire ocean basin is partitioned into square subgrids (as close as possible) that are allocated in the portion of the address space corresponding to the local memory of the individual processors, which are assigned responsibility for the subgrid. For the measurements in this chapter we use an input that has 130×130 grid points. There are five steps in a time iteration. Since data are exchanged between the steps, all the processors present synchronize at the end of each step before proceeding to the next. Communication occurs when the boundary points of a subgrid are accessed by the adjacent subgrid in nearest-neighbor fashion.

Computation/Communication for the Parallel Programs

A key characteristic in determining the performance of parallel programs is the ratio of computation to communication. If the ratio is high, it means the application has lots of computation for each datum communicated. As we saw in section 8.1, communication is the costly part of parallel computing; therefore high computation-to-communication ratios are very beneficial. In a parallel processing environment, we are concerned with how the ratio of computation to communication changes as we increase either the number of processors, the size of the problem, or both. Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up. Because we are often interested in running larger problems, it is vital to understand how changing the data set size affects this ratio.

To understand what happens quantitatively to the computation-to-communication ratio as we add processors, consider what happens separately to computation and to communication as we either add processors or increase problem size. For these applications Figure 8.4 shows that as we add processors, the amount of computation per processor falls proportionately and the amount of communication per processor falls more slowly. As we increase the problem size, the computation scales as the O( ) complexity of the algorithm dictates. Communication scaling is more complex and depends on details of the algorithm; we describe the basic phenomena for each application in the caption of Figure 8.4.

The overall computation-to-communication ratio is computed from the individual growth rate in computation and communication. In general, this rate rises slowly with an increase in data set size and decreases as we add processors. This reminds us that performing a fixed-size problem with more processors leads to increasing inefficiencies because the amount of communication among processors grows. It also tells us how quickly we must scale data set size as we add processors, to keep the fraction of time in communication fixed.

Multiprogramming and OS Workload

For small-scale multiprocessors we will also look at a multiprogrammed workload consisting of both user activity and OS activity. The workload used is two independent copies of the compile phase of the Andrew benchmark. The compile phase consists of a parallel make using eight processors. The workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems. The workload is run with 128 MB of memory, and no paging activity takes place.

The workload has three distinct phases: compiling the benchmarks, which involves substantial compute activity; installing the object files in a library; and removing the object files. The last phase is completely dominated by I/O and only two processes are active (one for each of the runs). In the middle phase, I/O also plays a major role and the processes are largely idle.

Application	Scaling of computation	Scaling of communication	Scaling of computation-to-communication
FFT	$\frac{n \log n}{p}$	$\frac{n}{p}$	$\log n$
LU	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\frac{n \log n}{p}$	Approximately $\frac{\sqrt{n}(\log n)}{\sqrt{p}}$	Approximately $\frac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

FIGURE 8.4 Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel machines. In this table *p* is the increased processor count and *n* is the increased data set size. Scaling is on a per-processor basis. The computation scales up with *n* at the rate given by O( ) analysis and scales down linearly as *p* is increased. Communication scaling is more complex. In FFT all data points must interact, so communication increases with *n* and decreases with *p.* In LU and Ocean, communication is proportional to the boundary of a block, so it scales with data set size at a rate proportional to the side of a square with *n* points, namely $\sqrt{n}$; for the same reason communication in these two applications scales inversely to $\sqrt{p}$. Barnes has the most complex scaling properties. Because of the fall-off of interaction between bodies, the basic number of interactions among bodies, which require communication, scales as $\sqrt{n}$. An additional factor of log n is needed to maintain the relationships among the bodies. As processor count is increased, communication scales inversely to $\sqrt{p}$.

Because both idle time and instruction cache performance are important in this workload, we examine these two issues here, focusing on the data cache performance later in the chapter. For the workload measurements, we assume the following memory and I/O systems:

I/O system	Memory
Level 1 instruction cache	32K bytes, two-way set associative with a 64-byte block, one clock cycle hit time
Level 1 data cache	32K bytes, two-way set associative with a 32-byte block, one clock cycle hit time
Level 2 cache	1M bytes unified, two-way set associative with a 128-byte block, hit time 10 clock cycles
Main memory	Single memory on a bus with an access time of 100 clock cycles
Disk system	Fixed access latency of 3 ms (less than normal to reduce idle time)

Figure 8.5 shows how the execution time breaks down for the eight processors using the parameters just listed. Execution time is broken into four components: idle—execution in the kernel mode idle loop; user—execution in user code; synchronization—execution or waiting for synchronization variables; and kernel—execution in the OS that is neither idle nor in synchronization access.

Mode	% instructions executed	% execution time
Idle	69%	64%
User	27%	27%
Sync	1%	2%
Kernel	3%	7%

FIGURE 8.5 The distribution of execution time in the multiprogrammed parallel make workload. The high fraction of idle time is due to disk latency when only one of the eight processes is active. These data and the subsequent measurements for this workload were collected with the SimOS system [Rosenblum 1995]. The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University, using the SimOS simulation system.

Unlike the parallel scientific workload, this multiprogramming workload has a significant instruction cache performance loss, at least for the OS. The instruction cache miss rate in the OS for a 32-byte block size, two set-associative cache varies from 1.7% for a 32-KB cache to 0.2% for a 256-KB cache. User-level, instruction cache misses are roughly one-sixth of the OS rate, across the variety of cache sizes.

8.3 Centralized Shared-Memory Architectures

Multis are a new class of computers based on multiple microprocessors. The small size, low cost, and high performance of microprocessors allow design and construction of computer structures that offer significant advantages in manufacture, price-performance ratio, and reliability over traditional computer families. ... Multis are likely to be the basis for the next, the fifth, generation of computers. [p. 463]

Bell [1985]

As we saw in Chapter 5, the use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor. If the main memory bandwidth demands of a single processor are reduced, multiple processors may be able to share the same memory. Starting in the 1980s, this observation, combined with the emerging dominance of the microprocessor, motivated many designers to create small-scale multiprocessors where several processors shared a single physical memory connected by a shared bus. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such machines are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists. Early designs of such machines were able to place an entire CPU and cache subsystem on a board, which plugged into the bus backplane. More recent designs have placed up to four processors per board; and by some time early in the next century, there may be multiple processors on a single die configured as a multiprocessor. Figure 8.1 on page 638 shows a simple diagram of such a machine.

The architecture supports the caching of both shared and private data. *Private data* is used by a single processor, while *shared data* is used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data, the program behavior is identical to that in a uniprocessor. When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces a new problem: cache coherence.

What Is Multiprocessor Cache Coherence?

As we saw in Chapter 6, the introduction of caches caused a coherence problem for I/O operations, since the view of memory through the cache could be different from the view of memory obtained through the I/O subsystem. The same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches. Figure 8.6 illustrates the problem and shows how two different processors can have two different values for the same location. This is generally referred to as the *cache-coherence* problem.

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

FIGURE 8.6 The cache-coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1!

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, while intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called *coherence*, defines what values can be returned by a read. The second aspect, called *consistency*, determines when a written value will be returned by a read. Let's look at coherence first.

A memory system is coherent if

1. A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated and no other writes to X occur between the two accesses.
3. Writes to the same location are serialized: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The first property simply preserves program order—we expect this property to be true even in uniprocessors. The second property defines the notion of what it means to have a coherent view of memory: If a processor could continuously read an old data value, we would clearly say that memory was incoherent.

The need for write serialization is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to serialize writes, so that all writes to the same location are seen in the same order; this property is called *write serialization*. Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important.

To understand why consistency is complex, observe that we cannot require that a read of X instantaneously see the value written for X by some other processor. If, for example, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly when a written value must be seen by a reader is defined by a *memory consistency model*—a topic discussed in section 8.6. Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations. For simplicity, and because we cannot explain the problem in full detail at this point, assume that we require that a write does not complete until all processors have seen the effect of the write and that the processor does not change the order of any write with any other memory access. This allows the processor to reorder reads, but forces the processor to finish a write in program order. We will rely on this assumption until we reach section 8.6, where we will see exactly the meaning of this definition, as well as the alternatives.

Basic Schemes for Enforcing Coherence

The coherence problem for multiprocessors and I/O, while similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will want to have copies of the same data in several caches. In a coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items. Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion; this reduces the latency to access a shared data item that is allocated remotely. Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item. Supporting this migration and replication is critical to performance in accessing shared data. Thus, rather than trying to solve the problem by avoiding it in software, small-scale multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

The protocols to maintain coherence for multiple processors are called *cache-coherence protocols*. Key to implementing a cache-coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols, which use different techniques to track the sharing status, in use:

- *Directory based*—The sharing status of a block of physical memory is kept in just one location, called the *directory;* we focus on this approach in section 8.4, when we discuss scalable shared-memory architecture.
- *Snooping*—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of a block that is requested on the bus. We focus on this approach in this section.

Snooping protocols became popular with multiprocessors using microprocessors and caches attached to a single shared memory because these protocols can use a preexisting physical connection—the bus to memory—to interrogate the status of the caches.

Alternative Protocols

There are two ways to maintain the coherence requirement described in the previous subsection. One method is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated. To see how this ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race (we'll see how we decide who wins shortly), causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization. Figure 8.7 shows an example of an invalidation protocol for a snooping bus with write-back caches in action.

Processor activity	**Bus activity**	**Contents of CPU A's cache**	**Contents of CPU B's cache**	**Contents of memory location X**
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 8.7 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This is typical in most protocols and simplifies the protocol, as we will see shortly.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *write broadcast* protocol. To keep the bandwidth requirements of this protocol under control it is useful to track whether or not a word in the cache is shared—that is, is contained in other caches. If it is not, then there is no need to broadcast or update any other caches. Figure 8.7 shows an example of a write update protocol in operation. In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs. To understand why, let's look at the qualitative performance differences.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

FIGURE 8.8 An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When CPU A broadcasts the write, both the cache in CPU B and the memory location of X are updated.

The performance differences between write update and write invalidate protocols arise from three characteristics:

1. Multiple writes to the same word with no intervening reads require multiple write broadcasts in an update protocol, but only one initial invalidation in a write invalidate protocol.
2. With multiword cache blocks, each word written in a cache block requires a write broadcast in an update protocol, while only the first write to any word in the block needs to generate an invalidate in an invalidation protocol. An invalidation protocol works on cache blocks, while an update protocol must work on individual words (or bytes, when bytes are written). It is possible to try to merge writes in a write broadcast scheme, just as we did for write buffers in Chapter 5, but the basic difference remains.
3. The delay between writing a word in one processor and reading the written value in another processor is usually less in a write update scheme, since the written data are immediately updated in the reader's cache (assuming that the reading processor has a copy of the data). By comparison, in an invalidation

protocol, the reader is invalidated first, then later reads the data and is stalled until a copy can be read and returned to the processor.

Because bus and memory bandwidth is usually the commodity most in demand in a bus-based multiprocessor, invalidation has become the protocol of choice for almost all implementations. Update protocols also cause problems for memory consistency models, reducing the potential performance gains of update, mentioned in point 3, even further. In designs with very small processor counts (2–4) where the processors are tightly coupled, the larger bandwidth demands of update may be acceptable. Nonetheless, given the trends in increasing processor performance and the related increase in bandwidth demands, we can expect update schemes to be used very infrequently. For this reason, we will focus only on invalidate protocols for the rest of the chapter.

Basic Implementation Techniques

The key to implementing an invalidate protocol in a small-scale machine is the use of the bus to perform invalidates. To perform an invalidate the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache is invalidated. The serialization of access enforced by the bus also forces serialization of writes, since when two processors compete to write to the same location, one must obtain bus access before the other. The first processor to obtain bus access will cause the other processor's copy to be invalidated, causing writes to be strictly serialized. One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched. (Write buffers can lead to some additional complexities, which are discussed in section 8.6.)

For a write-back cache, however, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for caches misses and for writes: Each processor snoops every address placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. Since write-back caches generate lower requirements for memory bandwidth, they are greatly preferable in a multiprocessor, despite the slight increase in complexity. Therefore, we focus on implementation with write-back caches.

The normal cache tags can be used to implement the process of snooping. Furthermore, the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward since they simply rely on the snooping capability. For writes we'd like to know whether any other copies of the block are cached, because, if there are no other cached copies, then the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

To track whether or not a cache block is shared we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as private. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared.

Since every bus transaction checks cache-address tags, this could potentially interfere with CPU cache accesses. This potential interference is reduced by one of two techniques: duplicating the tags or employing a multilevel cache with *inclusion,* whereby the levels closer to the CPU are a subset of those further away. If the tags are duplicated, then the CPU and the snooping activity may proceed in parallel. Of course, on a cache miss the processor needs to arbitrate for and update both sets of tags. Likewise, if the snoop finds a matching tag entry, it needs to arbitrate for and access both sets of cache tags (to perform an invalidate or to update the shared bit), as well as possibly the cache data array to retrieve a copy of a block. Thus with duplicate tags the processor only needs to be stalled when it does a cache access at the same time that a snoop has detected a copy in the cache. Furthermore, snooping activity is delayed only when the cache is dealing with a miss.

If the CPU uses a multilevel cache with the inclusion property, then every entry in the primary cache is also in the secondary cache. Thus the snoop activity can be directed to the second-level cache, while most of the processor's activity is directed to the primary cache. If the snoop gets a hit in the secondary cache, then it must arbitrate for the primary cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Since many multiprocessors use a multilevel cache to decrease the bandwidth demands of the individual processors, this solution has been adopted in many designs. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the CPU and the snooping activity. We discuss the inclusion property in more detail in section 8.8.

As you might imagine, there are many variations on cache coherence, depending on whether the scheme is invalidate based or update based, whether the cache is write back or write through, when updates occur, and if and how ownership is recorded. Figure 8.9 summarizes several snooping cache-coherence protocols and shows some machines that have used or are using that protocol.

Name	Protocol type	Memory-write policy	Unique feature	Machines using
Write Once	Write invalidate	Write back after first write	First snooping protocol described in literature	
Synapse N+1	Write invalidate	Write back	Explicit state where memory is the owner	Synapse machines; first cache-coherent machines available
Berkeley	Write invalidate	Write back	Owned shared state	Berkeley SPUR machine
Illinois	Write invalidate	Write back	Clean private state; can supply data from any cache with a clean copy	SGI Power and Challenge series
“Firefly”	Write broadcast	Write back when private, write through when shared	Memory updated on broadcast	No current machines; SPARCCenter 2000 closest.

FIGURE 8.9 Five snooping protocols summarized. Archibald and Baer [1986] use these names to describe the five protocols, and Eggers [1989] summarizes the similarities and differences as shown in this figure. The Firefly protocol was named for the experimental DEC Firefly multiprocessor, in which it appeared.

An Example Protocol

A bus-based coherence protocol is usually implemented by incorporating a finite state controller in each node. This controller responds to requests from the processor and from the bus, changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Figure 8.10 shows the requests

Request	Source	Function
Read hit	Processor	Read data in cache
Write hit	Processor	Write data in cache
Read miss	Bus	Request data from cache or memory
Write miss	Bus	Request data from cache or memory; perform any needed invalidates

FIGURE 8.10 The cache-coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request and the state of the cache block specified in the request.

generated by the processor-cache module in a node as well as those coming from the bus. For simplicity, the protocol we explain does not distinguish between a write hit and a write miss to a shared cache block: in both cases, we treat such an access as a write miss. When the write miss is placed on the bus, any processors with copies of the cache block invalidate it. In a write-back cache, if the block is exclusive in just one cache, that cache also writes back the block. Treating write hits to shared blocks as cache misses reduces the number of different bus transactions and simplifies the controller.

Figure 8.11 shows a finite-state transition diagram for a single cache block using a write-invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on CPU requests (on the left), as opposed to transitions based on bus requests (on the right). Boldface type is used to distinguish the bus actions, as opposed to the conditions on which a state transition depends. The state in each node represents the state of the selected cache block specified by the processor or bus request.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty states. All of the state changes indicated by arcs in the left half of Figure 8.11 would be needed in a write-back uniprocessor cache; the only difference in a multiprocessor with coherence is that the controller must generate a write miss when the controller has a write hit for a cache block in the shared state. The state changes represented by the arcs in the right half of Figure 8.11 are needed only for coherence and would not appear at all in a uniprocessor cache controller.

In reality, there is only one finite-state machine per cache, with stimuli coming either from the attached CPU or from the bus. Figure 8.12 shows how the state transitions in the right half of Figure 8.11 are combined with those in the left half of the figure to form a single state diagram for each cache block.

To understand why this protocol works, observe that any valid cache block is either in the shared state in multiple caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires a write miss to be placed on the bus, causing all caches to make the block invalid. In addition, if some other cache had the block in exclusive state, that cache generates a write back, which supplies the block containing the desired address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the owning cache also makes its state shared, forcing a subsequent write to require exclusive ownership. The actions in gray in Figure 8.12, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory. This simplifies the implementation, as we will see in detail in section 8.5.

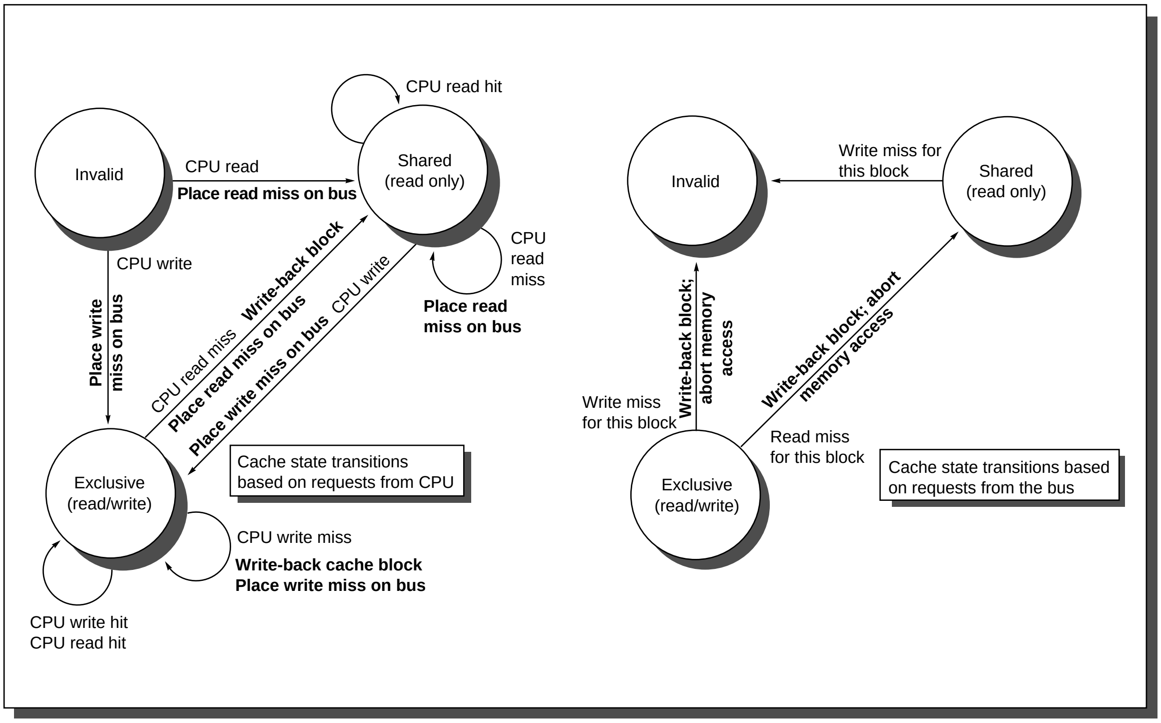


FIGURE 8.11 A write-invalidate, cache-coherence protocol for a write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles with any access permitted by the CPU without a state transition shown in parenthesis under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a line in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the CPU associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the CPU does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state always generates a miss, even if the block is present in the cache, since the block must be made exclusive. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined. This protocol is somewhat simpler than those in use in existing multiprocessors.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality this is not true. Similarly, if we used a split transaction bus (see Chapter 6, section 6.3), then read misses would also not be atomic.

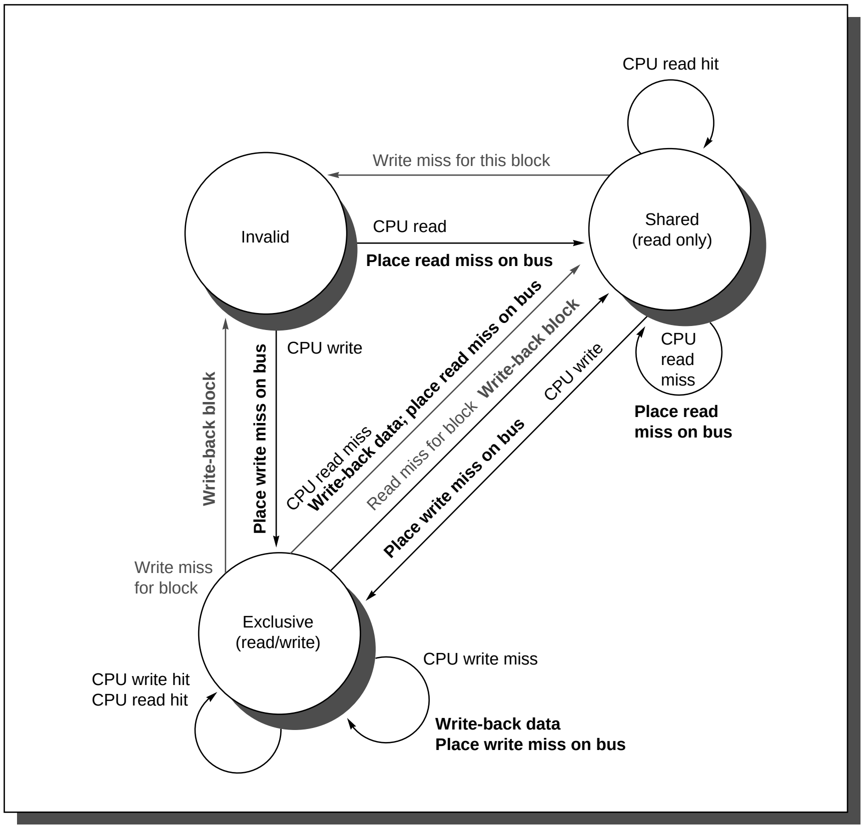


FIGURE 8.12 Cache-coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 8.11, the activities on a transition are shown in bold.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. Appendix E deals with these complex issues, showing how the protocol can be modified to deal with nonatomic writes without introducing deadlock.

As stated earlier, this coherence protocol is actually simpler than those used in practice. There are two major simplifications. First, in this protocol all transitions to the exclusive state generate a write miss on the bus, and we assume that the requesting cache always fills the block with the contents returned. This simplifies the detailed implementation. Most real protocols distinguish between a write miss and a write hit, which can occur when the cache block is initially in the shared state. Such misses are called *ownership* or *upgrade* misses, since they involve changing the state of the block, but do not actually require a data fetch. To support such state changes, the protocol uses an *invalidate operation,* in addition to a write miss. With such operations, however, the actual implementation of the protocol becomes slightly more complex.

The second major simplification is that many machines distinguish between a cache block that is really shared and one that exists in the clean state in exactly one cache. This addition of a “clean and private” state eliminates the need to generate a bus transaction on a write to such a block. Another enhancement in wide use allows other caches to supply data on a miss to a shared block. The next part of this section examines the performance of these protocols for our parallel and multiprogrammed workloads.

Performance of Snooping Coherence Protocols

In a bus-based multiprocessor using an invalidation protocol, several different phenomena combine to determine performance. In particular, the overall cache performance is a combination of the behavior of uniprocessor cache miss traffic and the traffic caused by communication, which results in invalidations and subsequent cache misses. Changing the processor count, cache size, and block size can affect these two components of the miss rate in different ways, leading to overall system behavior that is a combination of the two effects.

Performance for the Parallel Program Workload

In this section, we use a simulator to study the performance of our four parallel programs. For these measurements, the problem sizes are as follows:

- *Barnes-Hut*—16K bodies run for six time steps (the accuracy control is set to 1.0, a typical, realistic value);
- *FFT*—1 million complex data points
- *LU*—A 512×512 matrix is used with 16×16 blocks
- *Ocean*—A 130×130 grid with a typical error tolerance

In looking at the miss rates as we vary processor count, cache size, and block size, we decompose the total miss rate into *coherence misses* and normal uniprocessor misses. The normal uniprocessor misses consist of capacity, conflict, and compulsory misses. We label these misses as capacity misses, because that is the dominant cause for these benchmarks. For these measurements, we include as a coherence miss any write misses needed to upgrade a block from shared to exclusive, even though no one is sharing the cache block. This reflects a protocol that does not distinguish between a private and shared cache block.

Figure 8.13 shows the data miss rates for our four applications, as we increase the number of processors from one to 16, while keeping the problem size constant. As we increase the number of processors, the total amount of cache increases, usually causing the capacity misses to drop. In contrast, increasing the processor count usually causes the amount of communication to increase, in turn causing the coherence misses to rise. The magnitude of these two effects differs by application.

In FFT, the capacity miss rate drops (from nearly 7% to just over 5%) but the coherence miss rate increases (from about 1% to about 2.7%), leading to a constant overall miss rate. Ocean shows a combination of effects, including some that relate to the partitioning of the grid and how grid boundaries map to cache blocks. For a typical 2D grid code the communication-generated misses are proportional to the boundary of each partition of the grid, while the capacity misses are proportional to the area of the grid. Therefore, increasing the total amount of cache while keeping the total problem size fixed will have a more significant effect on the capacity miss rate, at least until each subgrid fits within an individual processor's cache. The significant jump in miss rate between one and two processors occurs because of conflicts that arise from the way in which the multiple grids are mapped to the caches. This conflict is present for direct-mapped and two-way set associative caches, but fades at higher associativities. Such conflicts are not unusual in array-based applications, especially when there are multiple grids in use at once. In Barnes and LU the increase in processor count has little effect on the miss rate, sometimes causing a slight increase and sometimes causing a slight decrease.

Increasing the cache size has a beneficial effect on performance, since it reduces the frequency of costly cache misses. Figure 8.14 illustrates the change in miss rate as cache size is increased, showing the portion of the miss rate due to coherence misses and to uniprocessor capacity misses. Two effects can lead to a miss rate that does not decrease—at least not as quickly as we might expect—as cache size increases: inherent communication and plateaus in the miss rate. Inherent communication leads to a certain frequency of coherence misses that are not significantly affected by increasing cache size. Thus if the cache size is increased while maintaining a fixed problem size, the coherence miss rate eventually limits the decrease in cache miss rate. This effect is most obvious in Barnes, where the coherence miss rate essentially becomes the entire miss rate.

A less important effect is a temporary plateau in the capacity miss rate that arises when the application has some fraction of its data present in cache but some significant portion of the data set does not fit in the cache or in caches that are slightly bigger. In LU, a very small cache (about 4 KB) can capture the pair of 16×16 blocks used in the inner loop; beyond that the next big improvement in capacity miss rate occurs when both matrices fit in the caches, which occurs when the total cache size is between 4 MB and 8 MB, a data point we will see later. This *working set effect* is partly at work between 32 KB and 128 KB for FFT, where the capacity miss rate drops only 0.3%. Beyond that cache size, a faster decrease in the capacity miss rate is seen, as some other major data structure be-

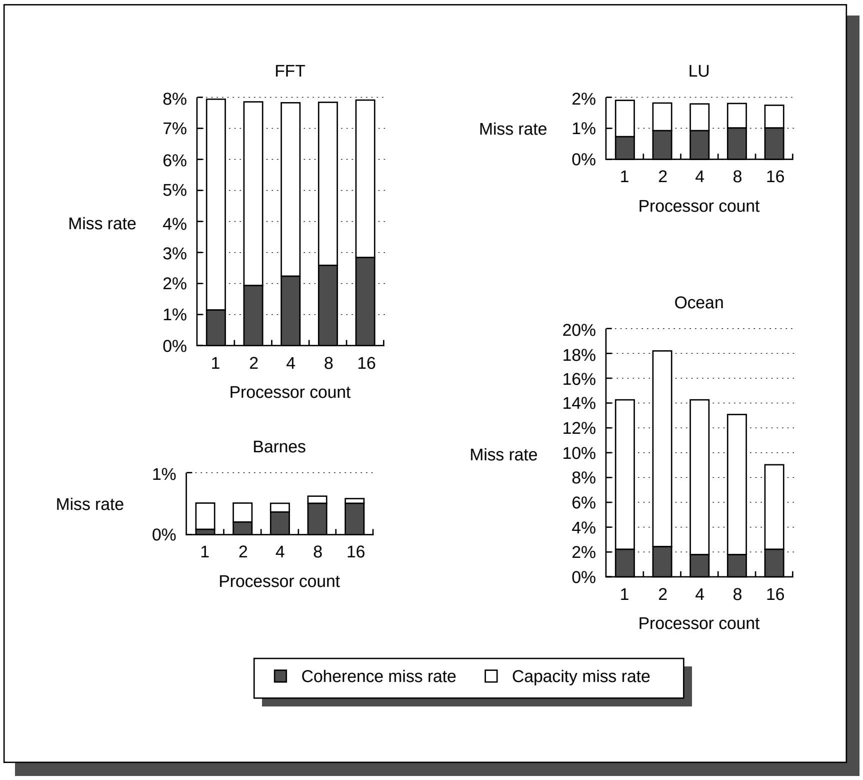


FIGURE 8.13 Data miss rates can vary in nonobvious ways as the processor count is increased from one to 16. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that is potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data is potentially shared if it is allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data is heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from one to two processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that is potentially, but not actually, shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single processor numbers include write upgrade misses, which occur in this protocol even if the data is not actually shared, since it is in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32 blocks. Notice that the scale for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.

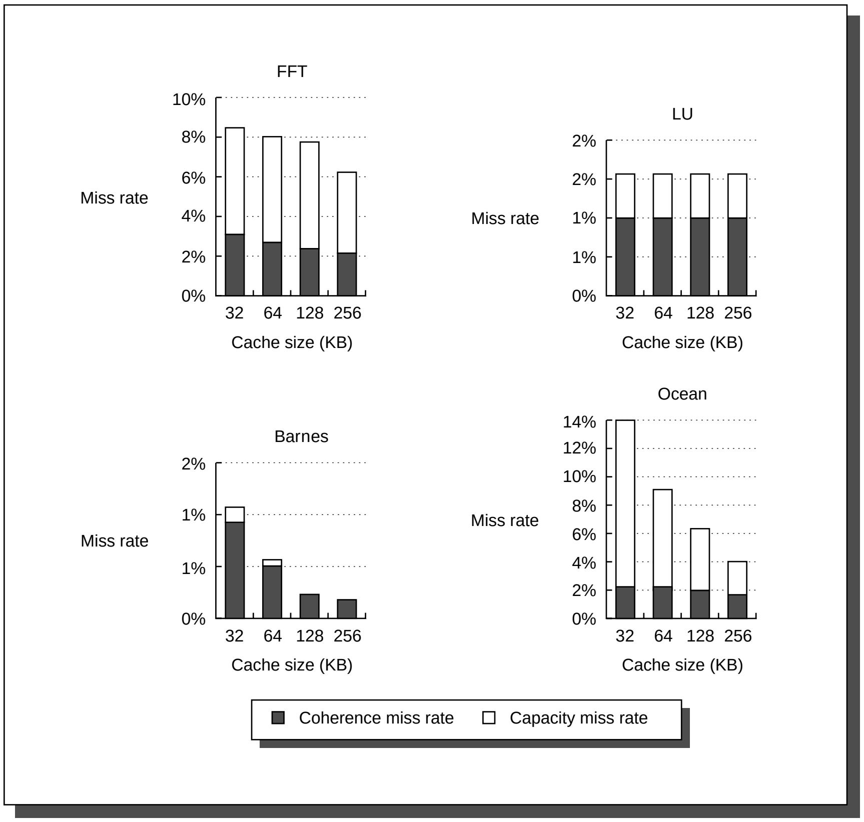


FIGURE 8.14 The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect. The block size is 32 bytes and the cache is two-way set-associative. The processor count is fixed at 16 processors. Observe that the scale for each graph is different.

gins to reside in the cache. These plateaus are common in programs that deal with large arrays in a structured fashion.

Increasing the block size is another way to change the miss rate in a cache. In uniprocessors, larger block sizes are often optimal with larger caches. In multiprocessors, two new effects come into play: a reduction in spatial locality for shared data and an effect called *false sharing*. Several studies have shown that shared data have lower spatial locality than unshared data. This means that for shared data, fetching larger blocks is less effective than in a uniprocessor, because the probability is higher that the block will be replaced before all its contents are referenced.

The second effect, false sharing, arises from the use of an invalidation-based coherence algorithm with a single valid bit per block. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. If the word written into is actually used by the processor that received the invalidate, then the reference was a true sharing reference and would have caused a miss independent of the block size or position of words. If, however, the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss, then it is a false sharing miss. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word. The following Example makes the sharing patterns clear.

EXAMPLE Assume that words x1 and x2 are in the same cache block in a clean state in the caches of P1 and P2, which have previously read both x1 and x2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

ANSWER Here are classifications by time step:

1. This event is a true sharing miss, since x1 was read by P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since x2 was invalidated by the write of x1 in P1, but that value of x1 is not used in P2.
3. This event is a false sharing miss, since the block containing x1 is marked shared due to the read in P2, but P2 did not read x1.
4. This event is a false sharing miss for the same reason as step 3.
5. This event is a true sharing miss, since the value being read was written by P2. ■

Figure 8.15 shows the miss rates as the cache block size is increased for a 16-processor run with a 64-KB cache. The most interesting behavior is in Barnes, where the miss rate initially declines and then rises due to an increase in the number of coherence misses, which probably occurs because of false sharing. In the other benchmarks, increasing the block size decreases the overall miss rate. In Ocean and LU, the block size increase affects both the coherence and capacity miss rates about equally. In FFT, the coherence miss rate is actually decreased at a faster rate than the capacity miss rate. This is because the communication in FFT is structured to be very efficient. In less optimized programs, we would expect more false sharing and less spatial locality for shared data, resulting in more behavior like that of Barnes.

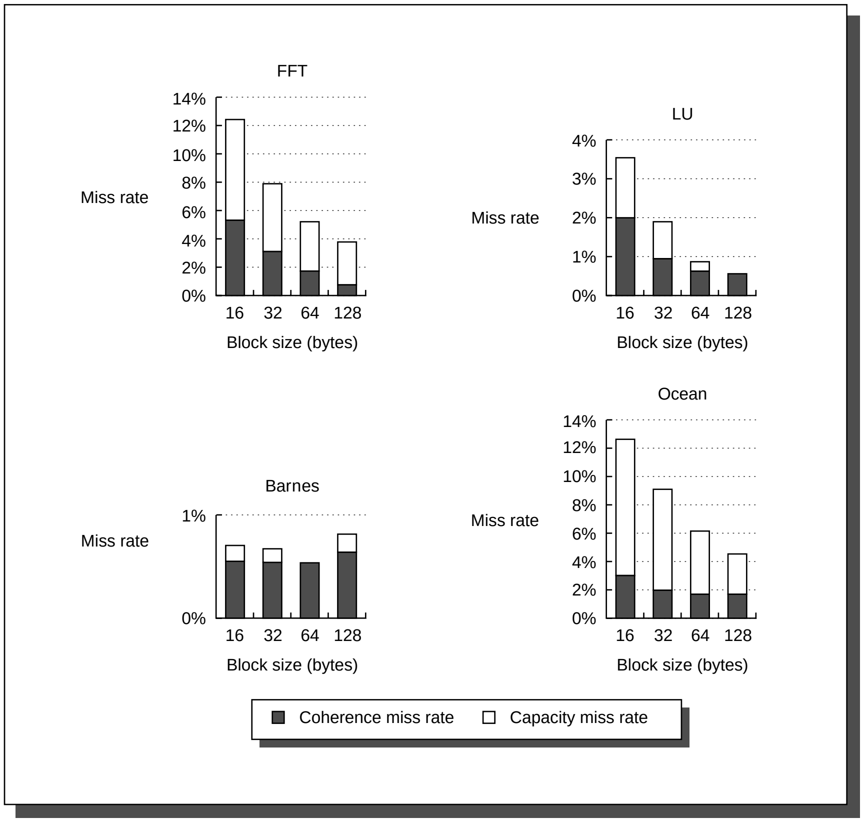


FIGURE 8.15 The data miss rate drops as the cache block size is increased. All these results are for a 16-processor run with a 64-KB cache and two-way set associativity. Once again we use different scales for each benchmark.

Although the drop in miss rates with longer blocks may lead you to believe that choosing a longer block size is the best decision, the bottleneck in bus-based multiprocessors is often the limited memory and bus bandwidth. Larger blocks mean more bytes on the bus per miss. Figure 8.16 shows the growth in bus traffic as the block size is increased. This growth is most serious in the programs that have a high miss rate, especially Ocean. The growth in traffic can actually lead to performance slowdowns due both to longer miss penalties and to increased bus contention.

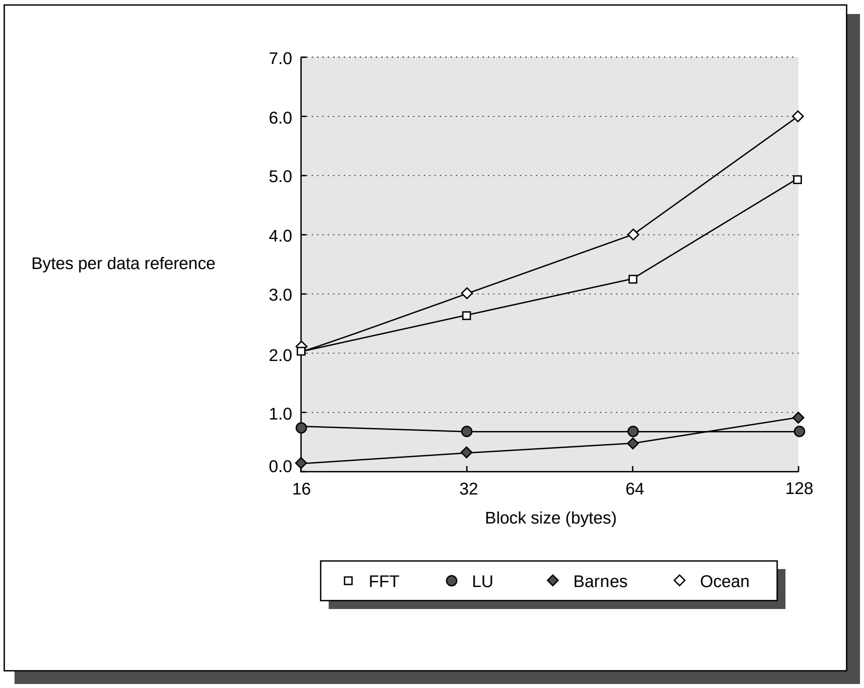


FIGURE 8.16 Bus traffic for data misses climbs steadily as the block size in the data cache is increased. The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes. Remember that our protocol treats ownership misses the same as other misses, slightly increasing the penalty for large cache blocks; in both Ocean and FFT this effect accounts for less than 10% of the traffic.

Performance of the Multiprogramming and OS Workload

In this subsection we examine the cache performance of the multiprogrammed workload as the cache size and block size are changed. The workload remains the same as described in the previous section: two independent parallel makes, each using up to eight processors. Because of differences between the behavior of the kernel and that of the user processes, we keep these two components separate.

Remember, though, that the user processes execute more than eight times as many instructions, so that the overall miss rate is determined primarily by the miss rate in user code, which, as we will see, is often one-fifth of the kernel miss rate.

Figure 8.17 shows the data miss rate versus data cache size for the kernel and user components. The misses can be broken into three significant classes:

- Compulsory misses represent the first access to this block by this processor and are significant in this workload.
- Coherence misses represent misses due to invalidations.
- Normal capacity misses include misses caused by interference between the OS and the user process and between multiple user processes. Conflict misses are included in this category.

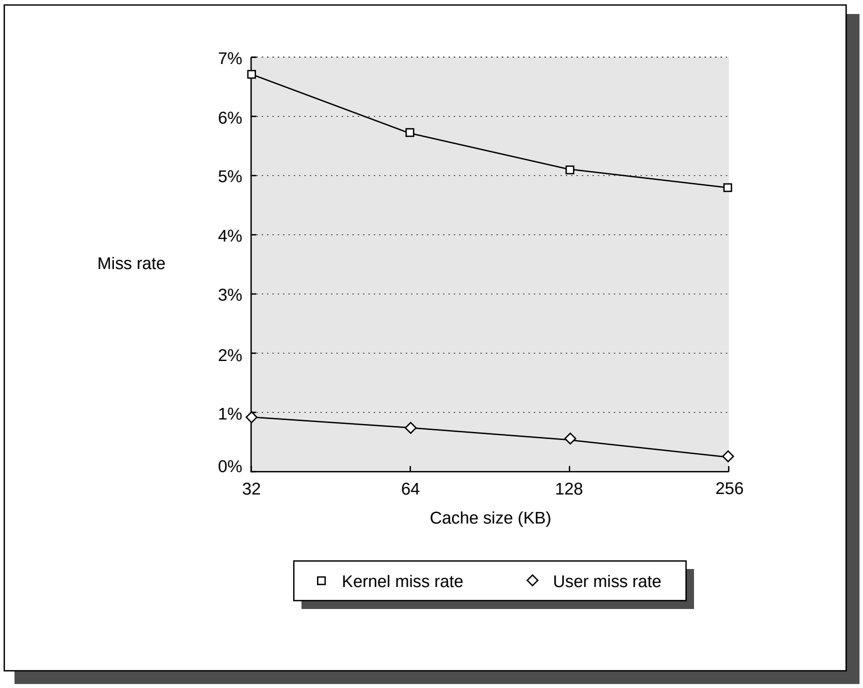


FIGURE 8.17 The data miss rate drops faster for the user code than for the kernel code as the data cache is increased from 32 KB to 256 KB with a 32-byte block. Although the user level miss rate drops by a factor of 3, the kernel level miss rate drops only by a factor of 1.3. As Figure 8.18 shows, this is due to a higher rate of compulsory misses and coherence misses.

For this workload the behavior of the operating system is more complex than the user processes. This is for two reasons. First, the kernel initializes all pages before allocating them to a user, which significantly increases the compulsory component of the kernel's miss rate. Second, the kernel actually shares data and thus has a nontrivial coherence miss rate. In contrast, user processes cause coherence misses only when the process is scheduled on a different processor; this component of the miss rate is small. Figure 8.18 shows the breakdown of the kernel miss rate as the cache size is increased.

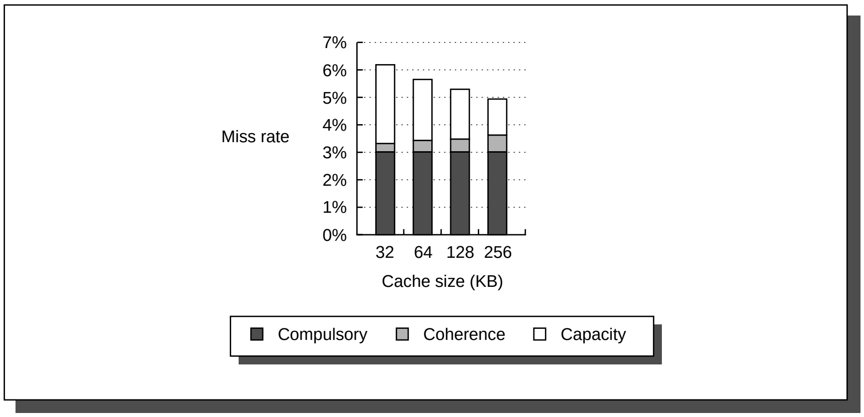


FIGURE 8.18 The components of the kernel data miss rate change as the data cache size is increased from 32KB to 256 KB. The compulsory miss rate component stays constant, since it is unaffected by cache size. The capacity component drops by more than a factor of two, while the coherence component nearly doubles. The increase in coherence misses occurs because the probability of a miss being caused by an invalidation increases with cache size, since fewer entries are bumped due to capacity.

Increasing the block size is likely to have more beneficial effects for this workload than for our parallel program workload, since a larger fraction of the misses arise from compulsory and capacity, both of which can be potentially improved with larger block sizes. Since coherence misses are relatively more rare, the negative effects of increasing block size should be small. Figure 8.19 shows how the miss rate for the kernel and user references changes as the block size is increased, assuming a 32 KB two-way set-associative data cache. Figure 8.20 confirms that, for the kernel references, the largest improvement is the reduction of the compulsory miss rate. As in the parallel programming workloads, the absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are insignificant.

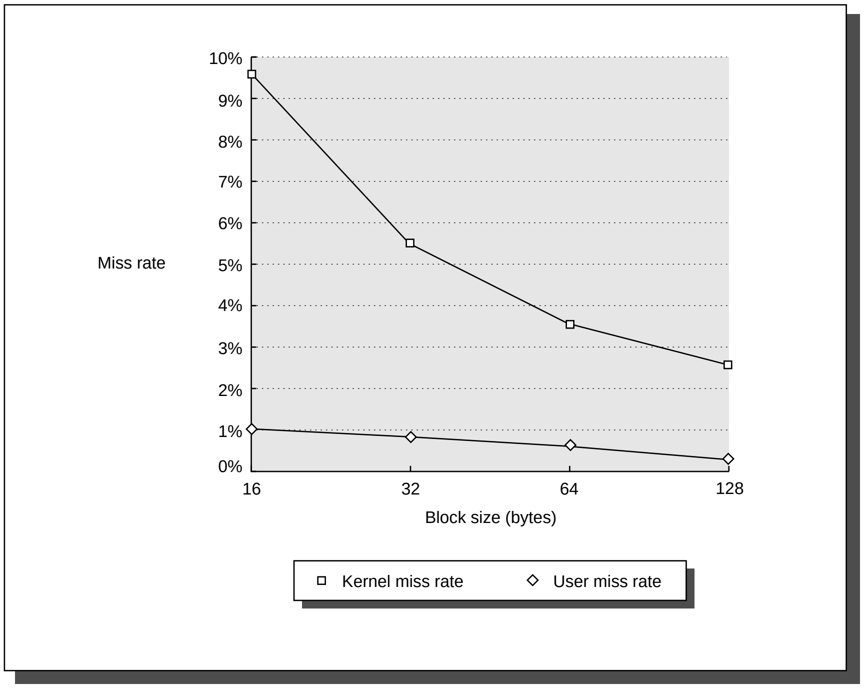


FIGURE 8.19 Miss rate drops steadily as the block size is increased for a 32-KB two-way set-associative data cache. As we might expect based on the higher compulsory component in the kernel, the improvement in miss rate for the kernel references is larger (almost a factor of 4 for the kernel references when going from 16-byte to 128-byte blocks versus just under a factor of 3 for the user references).

If we examine the number of bytes needed per data reference, as in Figure 8.21, we see that the behavior of the multiprogramming workload is like that of some programs in the parallel program workload. The kernel has a higher traffic ratio that grows quickly with block size. This is despite the significant reduction in compulsory misses; the smaller reduction in capacity and coherence misses drives an increase in total traffic. The user program has a much smaller traffic ratio that grows very slowly.

For the multiprogrammed workload, the OS is a much more demanding user of the memory system. If more OS or OS-like activity is included in the workload, it will become very difficult to build a sufficiently capable memory system.

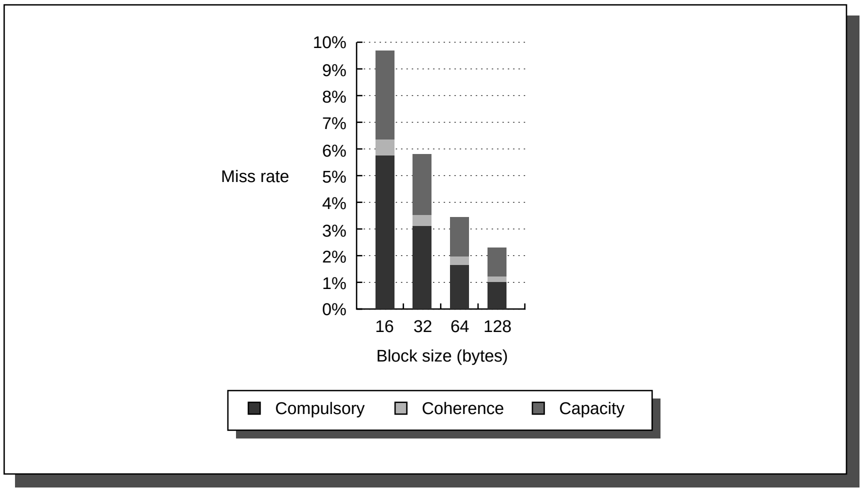


FIGURE 8.20 As we would expect, the increasing block size substantially reduces the compulsory miss rate in the kernel references. It also has a significant impact on the capacity miss rate, decreasing it by a factor of 2.4 over the range of block sizes. The increased block size has a small reduction in coherence traffic, which appears to stabilize at 64 bytes, with no change in the coherence miss rate in going to 128-byte lines. Because there are not significant reductions in the coherence miss rate as the block size increases, the fraction of the miss rate due to coherence grows from about 7% to about 15%.

Summary: Performance of Snooping Cache Schemes

In this section we examined the cache performance of both parallel program and multiprogrammed workloads. We saw that the coherence traffic can introduce new behaviors in the memory system that do not respond as easily to changes in cache size or block size that are normally used to improve uniprocessor cache performance. Coherence requests are a significant but not overwhelming component in the parallel processing workload. We can expect, however, that coherence requests will be more important in parallel programs that are less optimized.

In the multiprogrammed workload, the user and OS portions perform very differently, although neither has significant coherence traffic. In the OS portion, the compulsory and capacity contributions to the miss rate are much larger, leading to overall miss rates that are comparable to the worst programs in the parallel program workload. User cache performance, on the other hand, is very good and compares to the best programs in the parallel program workload.

The question of how these cache miss rates affect CPU performance depends on the rest of the memory system, including the latency and bandwidth of the bus and memory. We will return to overall performance in section 8.8, when we explore the design of the Challenge multiprocessor.

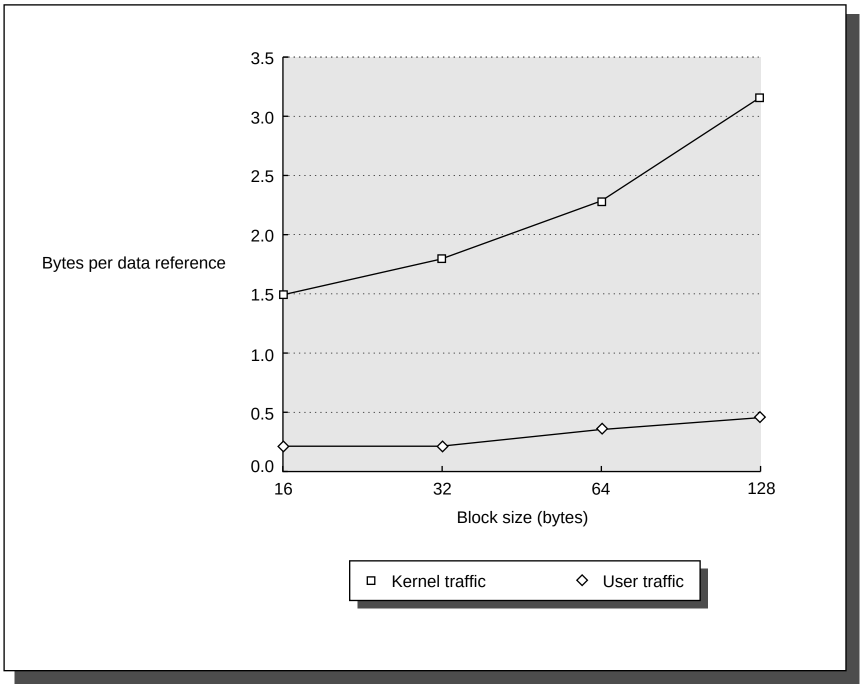


FIGURE 8.21 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart against the same chart for the parallel program workload shown in Figure 8.16.

8.4 Distributed Shared-Memory Architectures

A scalable machine supporting shared memory could choose to exclude or include cache coherence. The simplest scheme for the hardware is to exclude cache coherence, focusing instead on a scalable memory system. Several companies have built this style of machine; the Cray T3D is one well-known example. In such machines, memory is distributed among the nodes and all nodes are interconnected by a network. Access can be either local or remote—a controller inside each node decides, on the basis of the address, whether the data resides in the local memory or in a remote memory. In the latter case a message is sent to the controller in the remote memory to access the data.

These systems have caches, but to prevent coherence problems, shared data is marked as uncacheable and only private data is kept in the caches. Of course, software can still explicitly cache the value of shared data by copying the data from the shared portion of the address space to the local private portion of the address space that is cached. Coherence is then controlled by software. The advantage of such a mechanism is that little hardware support is required, although support for features such as block copy may be useful, since remote accesses fetch only single words (or double words) rather than cache blocks.

There are several major disadvantages to this approach. First, compiler mechanisms for transparent software cache coherence are very limited. The techniques that currently exist apply primarily to programs with well-structured loop-level parallelism, and these techniques have significant overhead arising from explicitly copying data. For irregular problems or problems involving dynamic data structures and pointers (including operating systems, for example), compiler-based software cache coherence is currently impractical. The basic difficulty is that software-based coherence algorithms must be conservative: every block that *might* be shared must be treated as if it *is* shared. This results in excess coherence overhead, because the compiler cannot predict the actual sharing accurately enough. Due to the complexity of the possible interactions, asking programmers to deal with coherence is unworkable.

Second, without cache coherence, the machine loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word. The benefits of spatial locality in shared data cannot be leveraged when single words are fetched from a remote memory for each reference. Support for a DMA mechanism among memories can help, but such mechanisms are often either costly to use (since they often require OS intervention) or expensive to implement since special-purpose hardware support and a buffer are needed. Furthermore, they are useful primarily when large block copies are needed (see Figure 7.25 on page 608 on the Cray T3D block copy).

Third, mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; we will examine this advantage in more detail later.

These disadvantages are magnified by the large latency of access to remote memory versus a local cache. For example, on the Cray T3D a local cache access has a latency of two cycles and is pipelined, while a remote access takes about 150 cycles.

For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors. For larger-scale architectures, there are new challenges to extending the cache-coherent shared-memory model. Although the bus can certainly be replaced with a more scalable interconnection network, and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed. A snooping protocol requires communication with all caches on every cache miss, including writes of potentially shared data. The absence of any centralized data structure that tracks the state of the caches is both the fundamental advantage of a snooping-based scheme, since it allows it to be inexpensive, as well as its Achilles' heel when it comes to scalability. For example, with only 16 processors and a block size of 64 bytes and a 64-KB data cache, the total bus bandwidth demand (ignoring stall cycles) for the four parallel programs in the workload ranges from almost 500 MB/sec (for Barnes) to over 9400 MB/sec (for Ocean), assuming a processor that issues a data reference every 5 ns, which is what a 1995 superscalar processor might generate. In comparison, the Silicon Graphics Challenge bus, the highest bandwidth bus-based multiprocessor in 1995, provides 1200 MB of bandwidth. Although the cache size used in these simulations is small, so is the problem size. Furthermore, although larger caches reduce the uniprocessor component of the traffic, they do not significantly reduce the parallel component of the miss rate.

Alternatively, we could build scalable shared-memory architectures that include cache coherency. The key is to find an alternative coherence protocol to the snooping protocol. One alternative protocol is a directory protocol. A directory keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on.

Existing directory implementations associate an entry in the directory with each memory block. In typical protocols, the amount of information is proportional to the product of the number of memory blocks and the number of processors. This is not a problem for machines with less than about a hundred processors, because the directory overhead will be tolerable. For larger machines, we need methods to allow the directory structure to be efficiently scaled. The methods that have been proposed either try to keep information for fewer blocks (e.g., only those in caches rather than all memory blocks) or try to keep fewer bits per entry.

To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property is what allows the coherence protocol to avoid broadcast. Figure 8.22 shows how our distributed-memory machine looks with the directories added to each node.

Directory-Based Cache-Coherence Protocols: The Basics

Just as with a snooping protocol, there are two primary operations that a directory protocol must implement: handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.) To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

- *Shared*—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches).
- *Uncached*—No processor has a copy of the cache block.

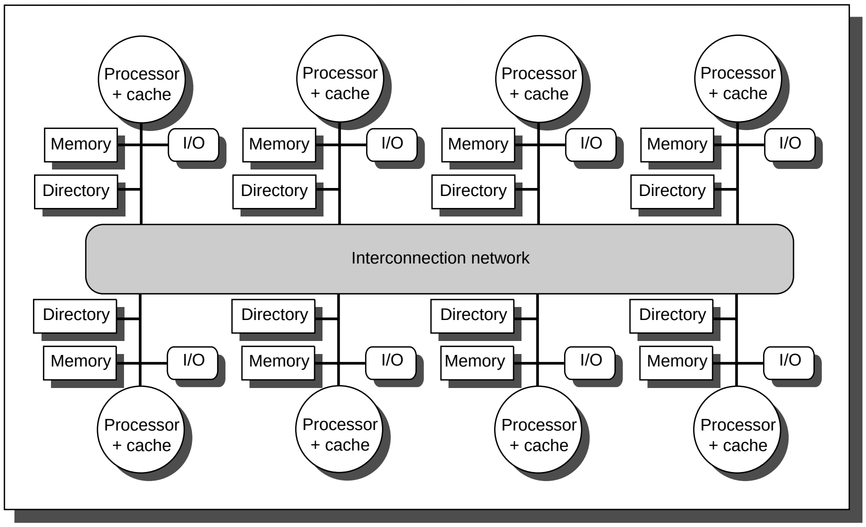


FIGURE 8.22 A directory is added to each node to implement cache coherence in a distributed-memory machine. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over a common bus, as shown, or it may have a separate port to memory, or it may be part of a central node controller through which all intranode and internode communications pass.

- *Exclusive*—Exactly one processor has a copy of the cache block and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.

In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. We make the same simplifying assumptions that we made in the case of the snooping cache: attempts to write data that is not exclusive in the writer's cache always generate write misses, and the processors block until an access completes. Since the interconnect is no longer a bus and we want to avoid broadcast, there are two additional complications. First, we cannot use the interconnect as a single point of arbitration, a function the bus performed in the snooping case. Second, because the interconnect is message oriented (unlike the bus, which is transaction oriented), many messages must have explicit responses.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories. Figure 8.23 shows the type of messages sent among nodes. The *local* node is the node where a request originates. The *home* node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node.

The *remote* node is the node that has a copy of a cache block, whether exclusive or shared. The local node may also be the home node, and vice versa. In either case the protocol is the same, although internode messages can be replaced by intranode transactions, which should be faster.

Message type	**Source**	**Destination**	**Message contents**	**Function of this message**
Read miss	Local cache	Home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Processor P has a write miss at address A; — request data and make P the exclusive owner.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	Data	Return a data value from the home memory.
Data write back	Remote cache	Home directory	A, data	Write back a data value for address A.

FIGURE 8.23 The possible messages sent among nodes to maintain coherence. The first two messages are miss requests sent by the local cache to the home. The third through fifth messages are messages sent to a remote cache by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol, since any dirty block must be exclusive and any shared block is always available in the home memory.

In this section, we assume a simple model of memory consistency. To minimize the type of messages and the complexity of the protocol, we make an assumption that messages will be received and acted upon in the same order they are sent. This assumption may not be true in practice, and can result in additional complications, some of which we address in section 8.6 when we discuss memory consistency models. In this section, we use this assumption to ensure that invalidates sent by a processor are honored immediately.

An Example Directory Protocol

The basic states of a cache block in a directory-based protocol are exactly like those in a snooping protocol, and the states in the directory are also analogous to those we showed earlier. Thus we can start with simple state diagrams that show the state transitions for an individual cache block and then examine the state diagram for the directory entry corresponding to each block in memory. As in the snooping case, these state transition diagrams do not represent all the details of a coherence protocol; however, the actual controller is highly dependent on a number of details of the machine (message delivery properties, buffering structures, and so on). In this section we present the basic protocol state diagrams. The knotty issues involved in implementing these state transition diagrams are examined in Appendix E, along with similar problems that arise for snooping caches.

Figure 8.24 shows the protocol actions to which an individual cache responds. We use the same notation as in the last section, with requests coming from outside the node in gray and actions in bold. The state transitions for an individual cache are caused by read misses, write misses, invalidates, and data fetch requests; these operations are all shown in Figure 8.24. An individual cache also generates read and write miss messages that are sent to the home directory. Read and write misses require data value replies, and these events wait for replies before changing state.

The operation of the state transition diagram for a cache block in Figure 8.24 is essentially the same as it is for the snooping case: the states are identical, and the stimulus is almost identical. The write miss operation, which was broadcast on the bus in the snooping scheme, is replaced by the data fetch and invalidate operations that are selectively sent by the directory controller. Like the snooping protocol, any cache block must be in the exclusive state when it is written and any shared block must be up to date in memory.

In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of actions: updates of the directory state, and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a block, but for all the cached copies of a memory block rather than for a single cache block. The memory block may be uncached by any node, cached in multiple nodes and readable (shared), or cached exclusively and writable in exactly

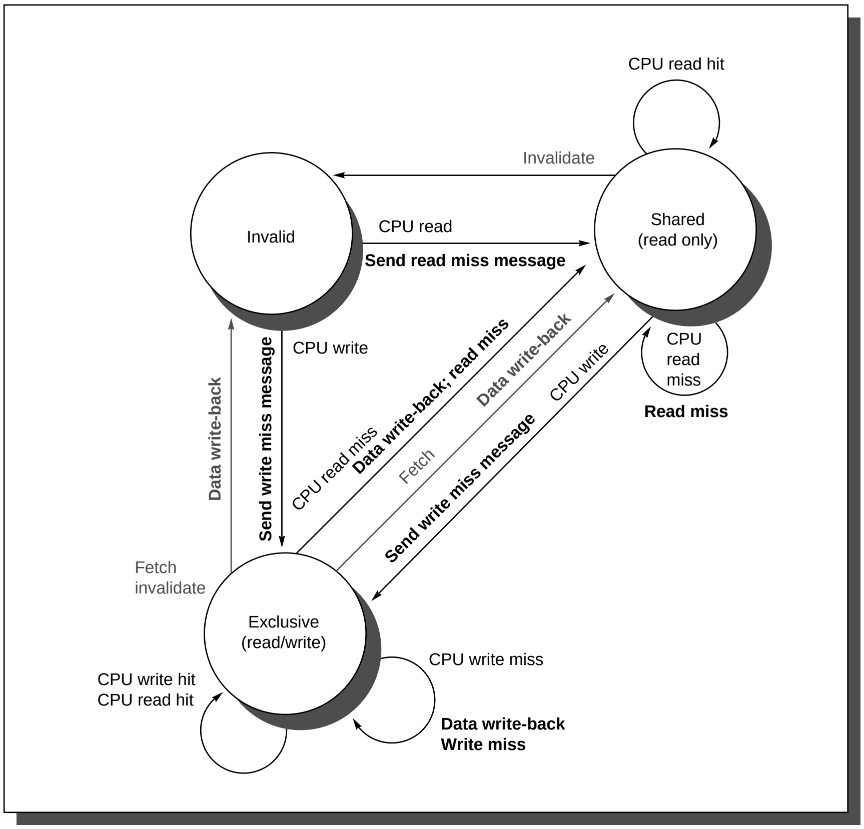


FIGURE 8.24 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in black and those from the home directory are shown in gray. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

one node. In addition to the state of each block, the directory must track the set of processors that have a copy of a block; we use a set called *Sharers* to perform this function. In small-scale machines (≤ 128 nodes), this set is typically kept as a bit vector. In larger machines, other techniques, which we discuss in the Exercises, are needed. Directory requests need to update the set Sharers and also read the set to perform invalidations.

Figure 8.25 shows the actions taken at the directory in response to messages received. The directory receives three different requests: read miss, write miss,

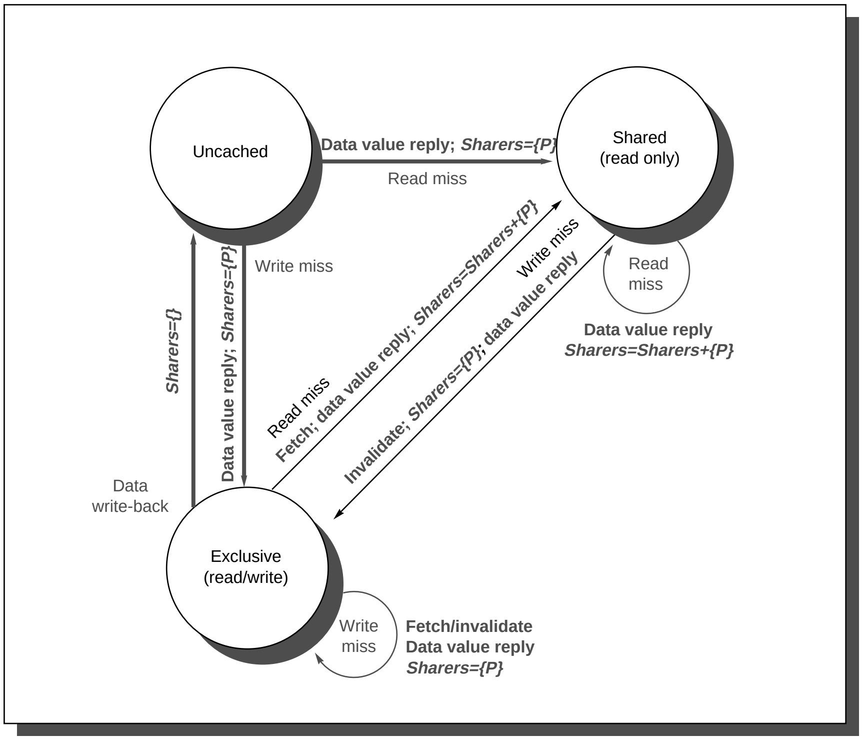


FIGURE 8.25 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request. Bold italics indicate an action that updates the sharing set, Sharers, as opposed to sending a message.

and data write back. The messages sent in response by the directory are shown in bold, while the updating of the set Sharers is shown in bold italics. Because all the stimulus messages are external, all actions are shown in gray. Our simplified protocol assumes that some actions are atomic, such as requesting a value and sending it to another node; a realistic implementation cannot use this assumption.

To understand these directory operations, let's examine the requests received and actions taken state by state. When a block is in the uncached state the copy in memory is the current value, so the only possible requests for that block are

- *Read miss*—The requesting processor is sent the requested data from memory and the requestor is made the only sharing node. The state of the block is made shared.

- *Write miss*—The requesting processor is sent the value and becomes the Sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state the memory value is up to date, so the same two requests can occur:

- *Read miss*—The requesting processor is sent the requested data from memory and the requesting processor is added to the sharing set.
- *Write miss*—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state the current value of the block is held in the cache of the processor identified by the set sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
- *Data write-back*—The owner processor is replacing the block and therefore must write it back. This makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the sharer set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner causing the cache to send the value of the block to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

This state transition diagram in Figure 8.25 is a simplification, just as it was in the snooping cache case. In the directory case it is a larger simplification, since our assumption that bus transactions are atomic no longer applies. Appendix E explores these issues in depth.

In addition, the directory protocols used in real machines contain additional optimizations. In particular, in our protocol here when a read or write miss occurs for a block that is exclusive, the block is first sent to the directory at the home node. From there it is stored into the home memory and also sent to the original requesting node. Many protocols in real machines forward the data from the owner node to the requesting node directly (as well as performing the write back to the home). Such optimizations may not add complexity to the protocol, but they often move the complexity from one part of the design to another.

Performance of Directory-Based Coherence Protocols

The performance of a directory-based machine depends on many of the same factors that influence the performance of bus-based machines (e.g., cache size, processor count, and block size), as well as the distribution of misses to various locations in the memory hierarchy. The location of a requested data item depends on both the initial allocation and the sharing patterns. We start by examining the basic cache performance of our parallel program workload and then look at the effect of different types of misses.

Because the machine is larger and has longer latencies than our snooping-based multiprocessor, we begin with a slightly larger cache (128 KB) and a block size of 64 bytes. In distributed memory architectures, the distribution of memory requests between local and remote is key to performance, because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section we separate the cache misses into local and remote requests. In looking at the figures, keep in mind that, for these applications, most of the remote misses that arise are coherence misses, although some capacity misses can also be remote, and in some applications with poor data distribution, such misses can be significant (see the Pitfall on page 738).

As Figure 8.26 shows, the miss rates with these cache sizes are not affected much by changes in processor count, with the exception of Ocean, where the miss rate rises at 64 processors. This rise occurs because of mapping conflicts in the cache that occur when the grid becomes small, leading to a rise in local misses, and because of a rise in the coherence misses, which are all remote.

Figure 8.27 shows how the miss rates change as the cache size is increased, assuming a 64-processor execution and 64-byte blocks. These miss rates decrease at rates that we might expect, although the dampening effect caused by little or no reduction in coherence misses leads to a slower decrease in the remote misses than in the local misses. By the time we reach the largest cache size shown, 512 KB, the remote miss rate is equal to or greater than the local miss rate. Larger caches would just continue to amplify this trend.

Finally, we examine the effect of changing the block size in Figure 8.28. Because these applications have good spatial locality, increases in block size reduce the miss rate, even for large blocks, although the performance benefits for going to the largest blocks are small. Furthermore, most of the improvement in miss rate comes in the local misses.

Rather than plot the memory traffic, Figure 8.29 plots the number of bytes required per data reference versus block size, breaking the requirement into local and global bandwidth. In the case of a bus, we can simply aggregate the demands of each processor to find the total demand for bus and memory bandwidth. For a scalable interconnect, we can use the data in Figure 8.29 to compute the required per-node global bandwidth and the estimated bisection bandwidth, as the next Example shows.

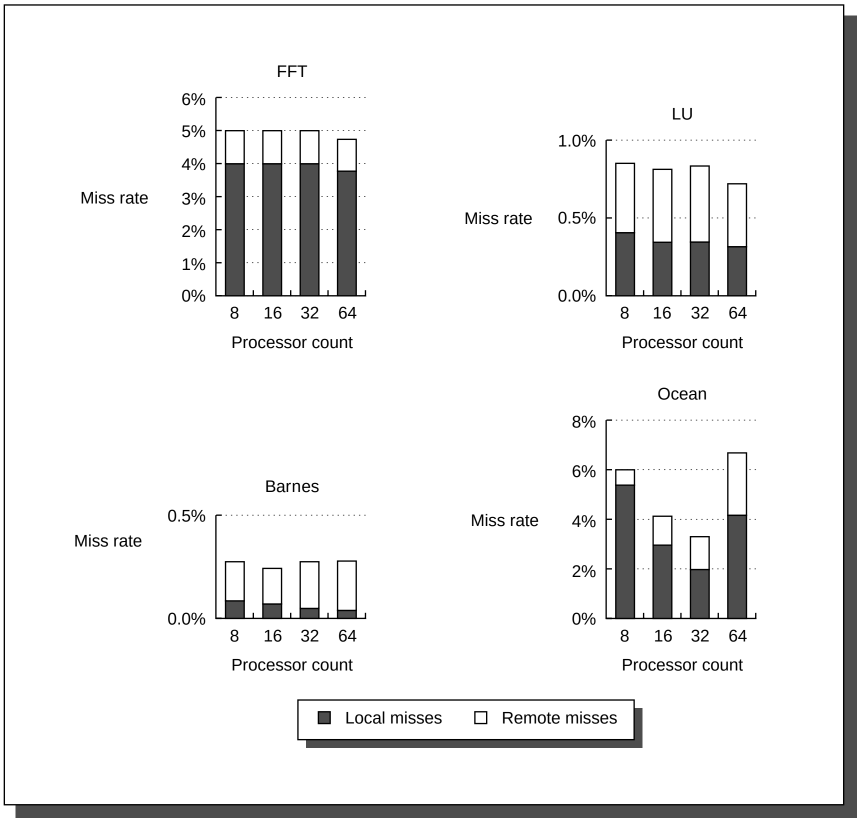


FIGURE 8.26 The data miss rate is often steady as processors are added for these benchmarks. Because of its grid structure, Ocean has an initially decreasing miss rate, which rises when there are 64 processors. For Ocean, the local miss rate drops from 5% at 8 processors to 2% at 32, before rising to 4% at 64. The remote miss rate in Ocean, driven primarily by communication, rises monotonically from 1% to 2.5%. Note that to show the detailed behavior of each benchmark, different scales are used on the y-axis. The cache for all these runs is 128 KB, two-way set associative, with 64-byte blocks. Remote misses include any misses that require communication with another node, whether to fetch the data or to deliver an invalidate. In particular, in this figure and other data in this section, the measurement of remote misses includes write upgrade misses where the data is up to date in the local memory but cached elsewhere and, therefore, requires invalidations to be sent. Such invalidations do indeed generate remote traffic, but may or may not delay the write, depending on the consistency model (see section 8.6).

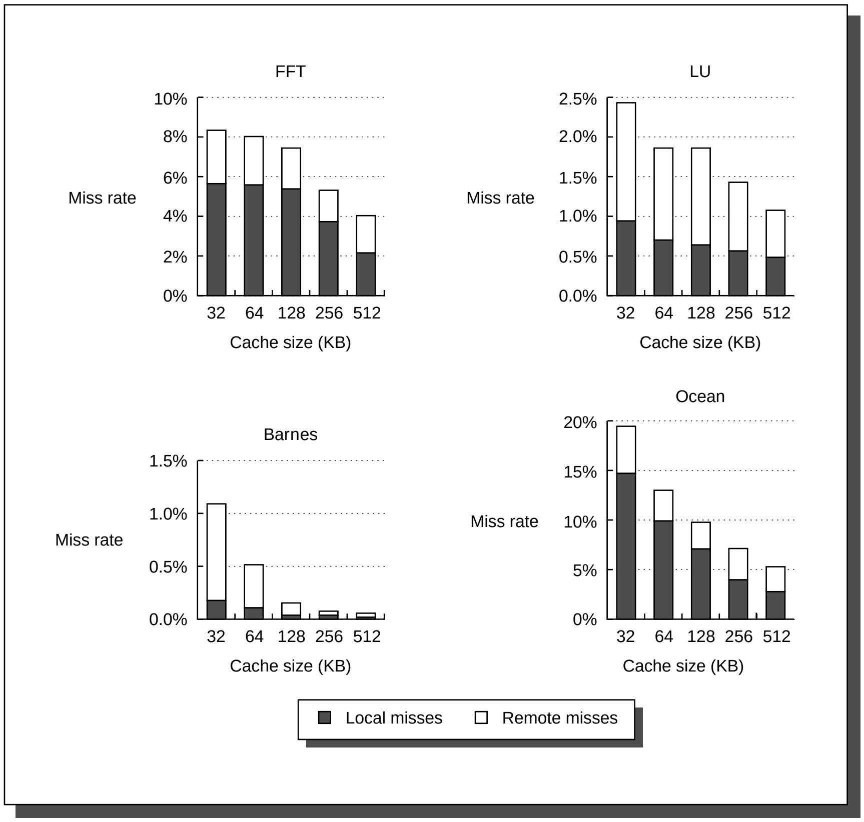


FIGURE 8.27 Miss rates decrease as cache sizes grow. Steady decreases are seen in the local miss rate, while the remote miss rate declines to varying degrees, depending on whether the remote miss rate had a large capacity component or was driven primarily by communication misses. In all cases, the decrease in the local miss rate is larger than the decrease in the remote miss rate. The plateau in the miss rate of FFT, which we mentioned in the last section, ends once the cache exceeds 128 KB. These runs were done with 64 processors and 64-byte cache blocks.

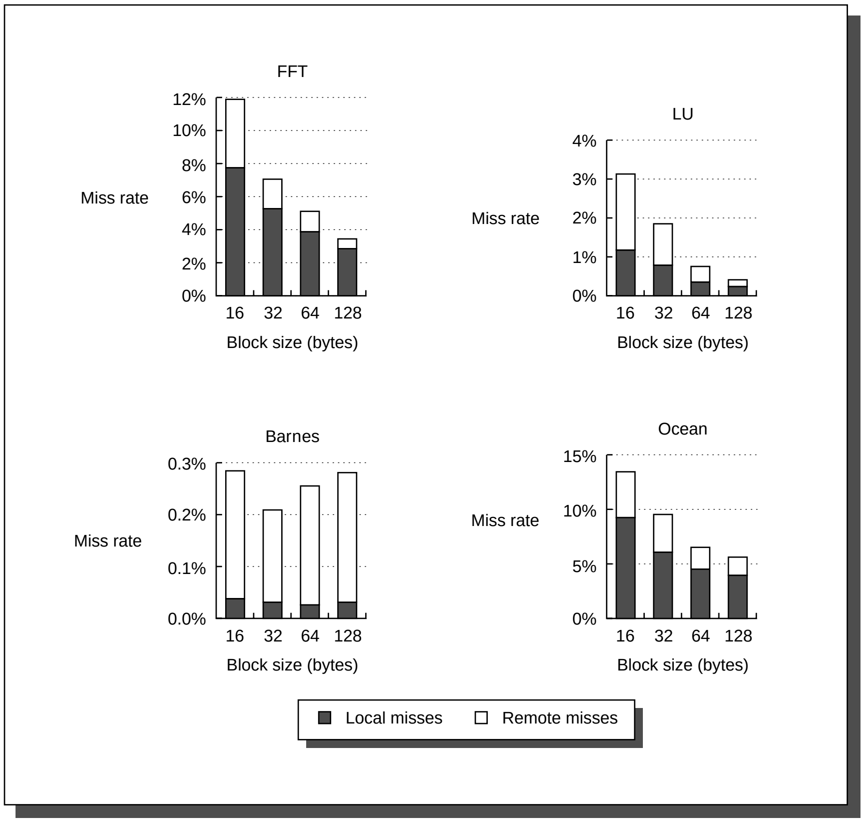


FIGURE 8.28 Data miss rate versus block size assuming a 128-KB cache and 64 processors in total. Although difficult to see, the coherence miss rate in Barnes actually rises for the largest block size, just as in the last section.

EXAMPLE Assume a multiprocessor with 64 200-MHz processors that sustains one memory reference per clock. For a 64-byte block size, the remote miss rate is 0.7%. Find the per-node and estimated bisection bandwidth for FFT. Assume that the processor does not stall for remote memory requests; this might be true if, for example, all remote data were prefetched. How do these bandwidth requirements compare to various interconnection technologies?

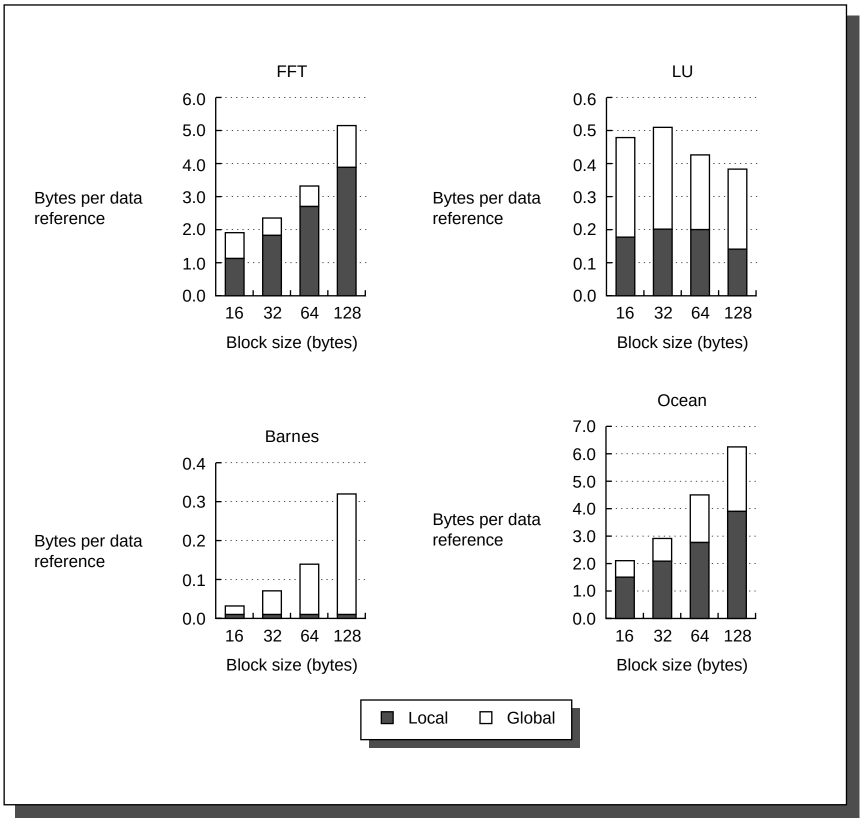


FIGURE 8.29 The number of bytes per data reference climbs steadily as block size is increased. These data can be used to determine the bandwidth required per node both internally and globally. The data assumes a 128-KB cache for each of 64 processors.

ANSWER The per-node bandwidth is simply the number of data bytes per reference times the reference rate: 0.7% × 200 × 64 = 90 MB/sec. This rate is about half the bandwidth of the fastest scalable MPP interconnects available in 1995. The FFT per-node bandwidth demand exceeds the fastest ATM interconnects available in 1995 by about a factor of 5, and slightly exceeds next-generation ATM.

FFT performs all-to-all communication, so the bisection bandwidth is equal to 32 times the per-node bandwidth, or 2880 MB/sec. For a 64-processor machine arranged in a 2D mesh, the bisection bandwidth grows as the square root of the number of processors. Thus for 64 processors the bisection bandwidth is 8 times the node bandwidth. In 1995, MPP-style interconnects offer about 200 MB/sec to a node for a total of 1600 MB/sec, or somewhat less than the required bandwidth. At 64 processors, a 3D mesh has double this bisection bandwidth (3200 MB/sec), which exceeds the required bandwidth. A next-generation 2D mesh is also expected to meet the bisection bandwidth requirement. A 1995 ATM-based 64×64 crossbar has about 1200 MB/sec of bisection bandwidth; a next-generation ATM offers four times this bandwidth, which exceeds the bisection bandwidth required, although it does not satisfy the per-node bandwidth. ■

The previous Example looked at the bandwidth demands. The other key issue for a parallel program is remote memory access time, or latency. To get insight into this, we use a simple example of a directory-based machine. Figure 8.30 shows the parameters we assume for our simple machine. It assumes that the time to first word for a local memory access is 25 cycles and that the path to local memory is 8 bytes wide, while the network interconnect is 2 bytes wide. This model ignores the effects of contention, which are probably not too serious in the parallel benchmarks we examine, with the possible exception of FFT, which uses all-to-all communication. Contention could have a serious performance impact in other work loads.

Characteristic	**Number of processor clock cycles**
Cache hit	1
Cache miss to local memory	$25 + \frac{\text{block size in bytes}}{8}$
Cache miss to remote home directory	$75 + \frac{\text{block size in bytes}}{2}$
Cache miss to remotely cached data (3-hop miss)	$100 + \frac{\text{block size in bytes}}{2}$

FIGURE 8.30 Characteristics of the example directory-based machine. Misses can be serviced locally (including from the local directory), at a remote home node, or using the services of both the home node and another remote node that is caching an exclusive copy. This last case is called a 3-hop miss and has a higher cost because it requires interrogating both the home directory and a remote cache. Note that this simple model does not account for invalidation time. These network latencies are typical of what can be achieved in 1995–96 in an MPP-style network interfaced in hardware to each node and assuming moderately fast processors (150–200 MHz).

Figure 8.31 shows the cost in cycles for the average memory reference, assuming the parameters in Figure 8.30. Only the latencies for each reference type are counted. Each bar indicates the contribution from cache hits, local misses, remote misses, and 3-hop remote misses. The cost is influenced by the total frequency of cache misses and upgrades, as well as by the distribution of the location where the miss is satisfied. The cost for a remote memory reference is fairly steady as the processor count is increased, except for Ocean. The increasing miss rate in Ocean for 64 processors is clear in Figure 8.26. As the miss rate increases, we should expect the time spent on memory references to increase also.

Although Figure 8.31 shows the memory access cost, which is the dominant multiprocessor cost in these benchmarks, a complete performance model would need to consider the effect of contention in the memory system, as well as the losses arising from synchronization delays. In section 8.8 we will look at the actual performance of the SGI Challenge system on these benchmarks.

The coherence protocols that we have discussed so far have made several simplifying assumptions. In practice, real protocols must deal with two realities: nonatomicity of operations and finite buffering. We have seen why certain operations (such as a write miss) cannot be atomic. In DSM machines the presence of only a finite number of buffers to hold message requests and replies introduces additional possibilities for deadlock. The challenge for the designer is to create a protocol that works correctly and without deadlock, using nonatomic actions and finite buffers as the building blocks. These factors are fundamental challenges in all parallel machines, and the solutions are applicable to a wide variety of protocol design environments, both in hardware and in software.

Because this material is extremely complex and not necessary to comprehend the rest of the chapter, we have placed it in Appendix E. For the interested reader, Appendix E shows how the specific problems in our coherence protocols are solved and illustrates the general principles that are more globally applicable. It describes the problems arising in snooping cache implementations, as well as the more complex problems that arise in more distributed systems using directories. If you want to understand how these machines really work and why designing them is such a challenge, go read Appendix E!

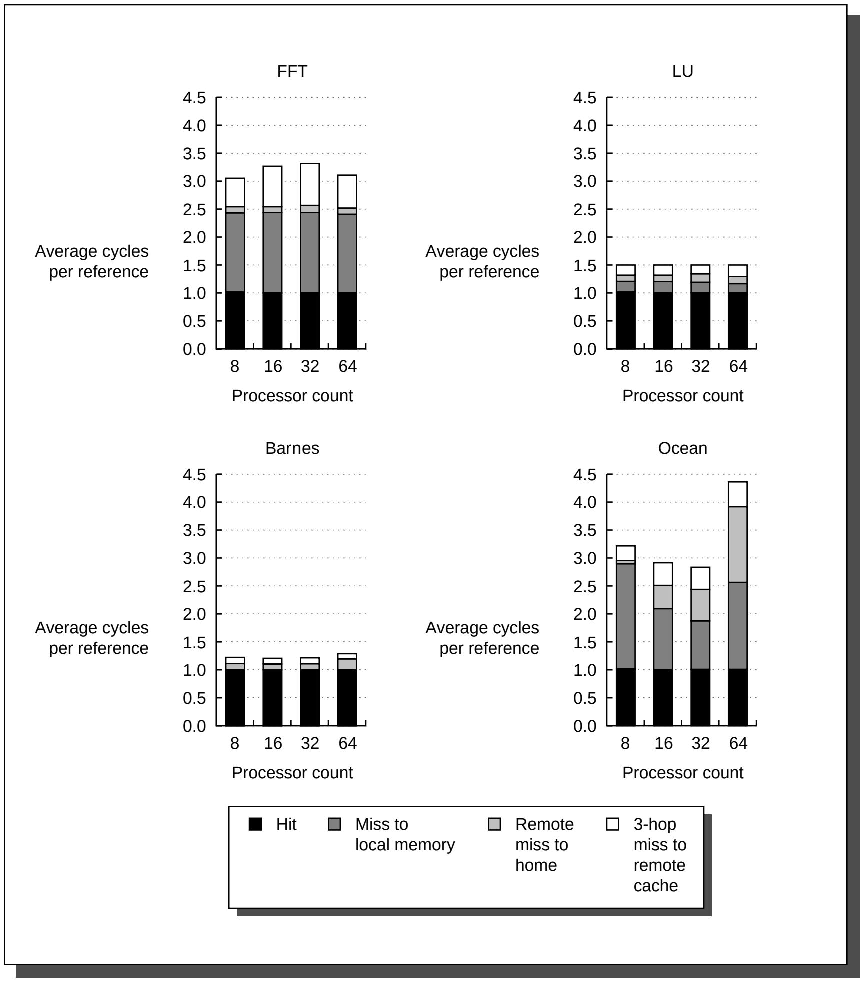


FIGURE 8.31 The effective latency of memory references in a DSM machine depends both on the relative frequency of cache misses and on the location of the memory where the accesses are served. These plots show the memory access cost (a metric called average memory access time in Chapter 5) for each of the benchmarks for 8, 16, 32, and 64 processors, assuming a 128-KB data cache that is two-way set associative with 64-byte blocks. The average memory access cost is composed of four different types of accesses, with the cost of each type given in Figure 8.30. For the Barnes and LU benchmarks, the low miss rates lead to low overall access times. In FFT, the higher access cost is determined by a higher local miss rate (4%) and a significant 3-hop miss rate (1%). Ocean shows the highest cost for memory accesses, as well as the only behavior that varies significantly with processor count. The high cost is driven primarily by a high local miss rate (average 1.4%). The memory access cost drops from 8 to 32 processors as the grids more easily fit in the individual caches. At 64 processors, the data set size is too small to map properly and both local misses and coherence misses rise, as we saw in Figure 8.26.

8.5 Synchronization

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. For smaller machines or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value. Software synchronization mechanisms are then constructed using this capability. For example, we will see how very efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism. In larger-scale machines or high-contention situations, synchronization can become a performance bottleneck, because contention introduces additional delays and because latency is potentially greater in such a machine. We will see how contention can arise in implementing some common user-level synchronization operations and examine more powerful hardware-supported synchronization primitives that can reduce contention as well as latency.

We begin by examining the basic hardware primitives, then construct several well-known synchronization routines with the primitives, and then turn to performance problems in larger machines and solutions for those problems.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases. There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky. Let's start with one such hardware primitive and show how it can be used to build some basic synchronization operations.

One typical operation for building synchronization operations is the *atomic exchan*ge, which interchanges a value in a register for a value in memory. To see how to use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: This race is broken since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible and two simultaneous exchanges will be ordered by the write serialization mechanisms. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically. One operation present in many older machines is *test-and-set,* which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange. Another atomic synchronization primitive is *fetch-and-increment:* it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment, which we will see shortly.

A slightly different approach to providing this atomic read-and-update operation has been used in some recent machines. Implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction. This complicates the implementation of coherence, since the hardware cannot allow any other operations between the read and the write, and yet must not deadlock.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic. The pair of instructions appears atomic if it appears as if all other operations executed by any processor appear before or after the pair. Thus when an instruction pair appears atomic, no other processor can change the value between the instruction pair.

The pair of instructions includes a special load called a *load linked* or *load locked* and a special store called a *store conditional*. These instructions are used in sequence: If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. If the processor does a context switch between the two instructions, then the store conditional also fails. The store conditional is defined to return a value indicating whether or not the store was successful. Since the load linked returns the initial value and the store conditional returns 1 if it succeeds and 0 otherwise, the following sequence implements an atomic exchange on the memory location specified by the contents of R1:

```
try:   MOV    R3,R4          ;mov exchange value
       LL     R2,0(R1)       ;load linked
       SC     R3,0(R1)       ;store conditional
       BEQZ   R3,try         ;branch store fails
       MOV    R4,R2          ;put load value in R4
```

At the end of this sequence the contents of R4 and the memory location specified by R1 have been atomically exchanged (ignoring any effect from delayed branches). Any time a processor intervenes and modifies the value in memory between the LL and SC instructions, the SC returns 0 in R3, causing the code sequence to try again.

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:   LL     R2,0(R1)       ;load linked
       ADDI   R3,R2,#1       ;increment
       SC     R3,0(R1)       ;store conditional
       BEQZ   R3,try         ;branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the LL instruction in a register, often called the *link register.* If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another SC), the link register is cleared. The SC instruction simply checks that its address matches that in the link register; if so, the SC succeeds; otherwise, it fails. Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the SC. In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

Implementing Locks Using Coherence

Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks:* locks that a processor continuously tries to acquire, spinning around a loop. Spin locks are used when we expect the lock to be held for a very short amount of time and when we want the process of locking to be low latency when the lock is available. Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
         LI     R2,#1
lockit:  EXCH   R2,0(R1)    ;atomic exchange
         BNEZ   R2,lockit   ;already locked?
```

If our machine supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. This has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

To obtain the first advantage—being able to spin on a local cached copy rather than generating a memory request for each attempt to acquire the lock—requires a change in our simple spin procedure. Each attempt to exchange in the loop directly above requires a write operation. If multiple processors are attempting to get the lock, each will generate the write. Most of these writes will lead to write misses, since each processor is trying to obtain the lock variable in an exclusive state.

Thus we should modify our spin-lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation. A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value of the read indicates that the lock is unlocked. The processor then races against all other processes that were similarly "spin waiting" to see who can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter.) The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again. Here is the code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit: LW     R2,0(R1)     ;load of lock
        BNEZ   R2,lockit    ;not available-spin
        LI     R2,#1        ;load locked value
        EXCH   R2,0(R1)     ;swap
        BNEZ   R2,lockit    ;branch if lock wasn’t 0
```

Let’s examine how this “spin-lock” scheme uses the cache-coherence mechanisms. Figure 8.32 shows the processor and bus or directory operations for multiple processes trying to lock a variable using an atomic swap. Once the processor with the lock stores a 0 into the lock, all other caches are invalidated and must fetch the new value to update their copy of the lock. One such cache gets the copy of the unlocked value (0) first and performs the swap. When the cache miss of other processors is satisfied, they find that the variable is already locked, so they must return to testing and spinning.

Step	Processor P0	Processor P1	Processor P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back from P0
4		(Waits while bus/ directory busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets Lock =1	Exclusive	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1	Enter critical section	Shared	Bus/directory services P1 cache miss; generates write back
8		Spins, testing if lock = 0			None

FIGURE 8.32 Cache-coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write-invalidate coherence. P0 starts with the lock (step 1). P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the critical section (steps 6 and 7), while P1’s attempt fails so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than eight clock ticks, since acquiring the bus and replying to misses takes much longer.

This example shows another advantage of the load-linked/store-conditional primitives: the read and write operation are explicitly separated. The load linked need not cause any bus traffic. This allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock):

```
lockit:  LL    R2,0(R1)     ;load linked
         BNEZ  R2,lockit    ;not available-spin
         LI    R2,#1        ;locked value
         SC    R2,0(R1)     ;store
         BEQZ  R2,lockit    ;branch if store fails
```

The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

Although our spin lock scheme is simple and compelling, it has difficulty scaling up to handle many processors because of the communication traffic generated when the lock is released. The next section discusses these problems in more detail, as well as techniques to overcome these problems in larger machines.

Synchronization Performance Challenges

To understand why the simple spin-lock scheme of the previous section does not scale well, imagine a large machine with all processors contending for the same lock. The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following Example shows how bad things can be.

EXAMPLE Suppose there are 20 processors on a bus that each try to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 50 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won't matter much!). Determine the number of bus transactions required for all 20 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 20 requests? Assume that the bus is totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

ANSWER Figure 8.33 shows the sequence of events from the time of the release to the time to the next release. Of course, the number of processors contending for the lock drops by one each time the lock is acquired, which reduces the average cost to 1525 cycles. Thus for 20 lock-unlock pairs it will take over 30,000 cycles for the processors to pass through the lock. Furthermore, the average processor will spend half this time idle, simply trying to get the lock. The number of bus transactions involved is over 400!

Event	**Duration**
Read miss by all waiting processors to fetch lock (20×50)	1000
Write miss by releasing processor and invalidates	50
Read miss by all waiting processors (20×50)	1000
Write miss by all waiting processors, one successful lock (50), and invalidation of all lock copies (19×50)	1000
Total time for one processor to acquire and release lock	3050 clocks

FIGURE 8.33 The time to acquire and release a single lock when 20 processors contend for the lock, assuming each bus transaction takes 50 clock cycles. Because of fair bus arbitration, the releasing processor must wait for *all* other 19 processors to try to get the lock in vain!

■

The difficulty in this Example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access. The fairness property of the bus actually makes things worse, since it delays the processor that claims the lock from releasing it; unfortunately, for any bus arbitration scheme some worst-case scenario does exist. The root of the problem is the contention and the fact that the lock access is serialized. The key advantages of spin locks, namely that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor, are both lost in this example. We will consider alternative implementations in the next section, but before we do that, let's consider the use of spin locks to implement another common high-level synchronization primitive.

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier.* A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier. To implement a barrier we usually use the ability to spin on a variable until it satisfies a test; we use the notation `spin(condition)` to indicate this. Figure 8.34 is a typical implementation, assuming that lock and unlock provide basic spin locks and `total` is the number of processes that must reach the barrier.

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count = count +1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=1;/* release processes */
}
else { /* more to come */

       spin (release==1);/* wait for arrivals */
}
```

FIGURE 8.34 Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable `release` is used to hold the processes until the last one reaches the barrier.The operation `spin (release==1)` causes a process to wait until all processes reach the barrier.

In practice, another complication makes barrier implementation slightly more complex. Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again. Assume that one of the processes never actually leaves the barrier (it stays at the spin operation), which could happen if the OS scheduled another process, for example. Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The fast process traps that last slow process in the barrier by resetting the flag `release`. Now all the processes will wait infinitely at the next instance of this barrier, because one process is trapped at the last instance, and the number of processes can never reach the value of total. The important observation is that the programmer did nothing wrong. Instead, the implementer of the barrier made some assumptions about forward progress that cannot be assumed. One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier. This would significantly increase the latency of the barrier and the contention, which as we will see shortly are already large. An alternative solution is a *sense-reversing barrier,* which makes use of a private per-process variable, `local_sense`, which is initialized to 1 for each process. Figure 8.34 shows the code for the sense-reversing barrier. This version of a barrier is safely usable; however, as the next example shows, its performance can still be quite poor.

```
local_sense = ! local_sense; /*toggle local_sense*/
lock (counterlock);/* ensure update atomic */
count=count+1;/* count arrivals */
unlock (counterlock);/* unlock */
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=local_sense;/* release processes */
}
else { /* more to come */
       spin (release==local_sense);/*wait for signal*/
}
```

FIGURE 8.35 Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag release, which controls the exit from the barrier. If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of `release` as it did in Figure 8.34.

EXAMPLE Suppose there are 20 processors on a bus that each try to execute a barrier simultaneously. Assume that each bus transaction is 50 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 20 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don't worry about counting the processors out of the barrier. How long will the entire process take?

ANSWER The following table shows the sequence of events for one processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock.

Event	Duration in clocks for one processor	Duration in clocks for 20 processors
Time for each processor to grab lock, increment, release lock	1525	30,500
Time to execute release	50	50
Time for each processor to get the release flag	50	1000
Total	1625	31,550

Our barrier operation takes a little longer than the 20-processor lock-unlock sequence we considered earlier. The total number of bus transactions is about 440. ■

As we can see from these examples, synchronization performance can be a real bottleneck when there is substantial contention among multiple processes. When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation. Our basic spin-lock operation can do this in two bus cycles: one to initially read the lock and one to write it. We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation. If the lock were almost always free, this could be better, but if the lock were not free, it would lead to lots of bus traffic, since each attempt to lock the variable would lead to a bus cycle. In practice, the latency of our spin lock is not quite as bad as we have seen in this example, since the write miss for a data item present in the cache is treated as an upgrade and will be cheaper than a true read miss.

The more serious problem in these examples is the serialization of each process's attempt to complete the synchronization. This serialization is a problem when there is contention, because it greatly increases the time to complete the synchronization operation. For example, if the time to complete all 20 lock and unlock operations depended only on the latency in the uncontended case, then it would take 2000 rather than 40,000 cycles to complete the synchronization operations. The use of a bus interconnect exacerbates this problem, but serialization could be just as serious in a directory-based machine, where the latency would be large. The next section presents some solutions that are useful when either the contention is high or the processor count is large.

Synchronization Mechanisms for Larger-Scale Machines

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant. We begin by showing how software implementations can improve the performance of locks and barriers when contention is high; we then explore two basic hardware primitives that reduce serialization while keeping latency low.

Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. This is done by delaying attempts to reacquire the lock whenever the store-conditional operation fails. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails. Figure 8.36 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses (see section 7.7). This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
         LI     R3,#1        ;R3 = initial delay
lockit:  LL     R2,0(R1)     ;load linked
         BNEZ   R2,lockit    ;not available-spin
         ADDI   R2,R2,#1     ;get locked value
         SC     R2,0(R1)     ;store conditional
         BNEZ   R2,gotit     ;branch if store succeeds
         SLL    R3,R3,#1     ;increase delay by factor of 2
         PAUSE  R3           ;delays by value in R3
         J      lockit
gotit:   use data protected by lock
```

FIGURE 8.36 A spin lock with exponential back-off. When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing R3 until it reaches 0. The exact timing of the delay is machine dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement `pause R3` should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again.

Another technique for implementing locks is to use queuing locks. We show how this works in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits (see Exercise 8.24). Before we look at hardware primitives, let's look at a better mechanism for barriers.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when all the processes must read the release flag. The former is more serious because it requires exclusive access to the synchronization variable and thus creates much more serialization; in comparison, the latter generates only read contention. We can reduce the contention by using a *combining tree,* a structure where multiple requests are locally combined in tree fashion. The same combining tree can be used to implement the release process, reducing the contention there; we leave the last step for the Exercises.

Our combining tree barrier uses a predetermined *n*-ary tree structure. We use the variable *k* to stand for the fan-in; in practice *k* = 4 seems to work well. When the *k*th process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes. As in our earlier example, we use a sense-reversing technique. The following tree-based barrier uses a tree to combine the processes and a single signal to release the barrier.

```
struct node{ /* a node in the combining tree */
    int counterlock; /* lock for this node */
    int count; /* counter for this node */
    int parent; /* parent in the tree = 0..P–1 except for root
                     = –1*/
};
struct node tree [0..P–1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode) {
    lock (tree[mynode].counterlock); /* protect count */
    tree[mynode].count=tree[mynode].count+1;
        /* increment count */
    unlock (tree[mynode].counterlock); /* unlock */
    if (tree[mynode].count==k) { /* all arrived at mynode */
        if (tree[mynode].parent >=0) {
            barrier(tree[mynode].parent);
        } else{
            release = local_sense;
        }
        tree[mynode].count = 0; /* reset for the next time */
    } else{
        spin (release==local_sense); /* wait */
    };
};
/* code executed by a processor to join barrier */
local_sense = ! local_sense;
barrier (mynode);
```

The tree is assumed to be prebuilt statically using the nodes in the array `tree`. Each node in the tree combines *k* processes and provides a separate counter and lock, so that at most *k* processes contend at each node. When the *k*th process reaches a node in the tree it goes up to the parent, incrementing the count at the parent. When the count in the parent node reaches *k*, the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier. Exercises 8.22 and 8.23 ask you to analyze the time for the combining barrier versus the noncombining version. Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but whether such facilities will be included in future machines is unclear.

Hardware Primitives

In this section we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This happens on each of the 20 lock/unlock sequences. We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, which we describe here, or in software using an array to keep track of the waiting processes. The basic concepts are the same in either case. Our hardware implementation assumes a directory-based machine where the individual processor caches are addressable. In a bus-based machine, a software implementation would be more appropriate and would have each processor using a different address for the lock, permitting the explicit transfer of the lock from one process to another.

How does a queuing lock work? On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable, the controller creates a record of the node's request (such as a bit in a vector) and sends the processor back a locked value for the variable, which the processor then spins on. When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor's cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

EXAMPLE How many bus transaction and how long does it take to have 20 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as before.

ANSWER Each processor misses once on the lock initially and once to free the lock, so it takes only 40 bus cycles. The first 20 initial misses take 1000 cycles, followed by a 50-cycle delay for each of the 20 releases. This is a total of 2050 cycles—significantly better than the case with conventional coherence-based spin locks. ■

There are a couple of key insights in implementing such a queuing lock capability. First, we need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release, so we can provide the lock to another processor. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based machine, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations. One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

Queuing locks can be used to improve the performance of our barrier operation (see Exercise 8.15). Alternatively, we can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks. One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment*, which atomically fetches a variable and increments its value. The returned value can be either the incremented value or the fetched value. Using fetch-and-increment we can dramatically improve our barrier implementation, compared to the simple code-sensing barrier.

EXAMPLE Write the code for the barrier using fetch-and-increment. Making the same assumptions as in our earlier example and also assuming that a fetch-and-increment operation takes 50 clock cycles, determine the time for 20 processors to traverse the barrier. How many bus cycles are required?

ANSWER Figure 8.37 shows the code for the barrier. This implementation requires 20 fetch-and-increment operations and 20 cache misses for the release operation. This is a total time of 2000 cycles and 40 bus/interconnect operations versus an earlier implementation that took over 15 times longer and 10 times more bus operations to complete the barrier. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree.

```
local_sense = ! local_sense; /*toggle local_sense*/
fetch_and_increment(count);/* atomic update*/
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=local_sense;/* release processes */
}
else { /* more to come */
       spin (release==local_sense);/*wait for signal*/
}
```

FIGURE 8.37 Code for a sense-reversing barrier using fetch-and-increment to do the counting.

■

As we have seen, synchronization problems can become quite acute in larger-scale machines. When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel machines is very challenging. In section 8.8 we will examine the costs of synchronization on an existing bus-based multiprocessor for some real applications.

8.6 Models of Memory Consistency

Cache coherence ensures that multiple processors see a consistent view of memory. It does not answer the question of *how* consistent the view of memory must be. By this we mean, When must a processor see a value that has been updated by another processor?

Since processors communicate through shared variables (both those for data values and those used for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor?

Since the only way to “observe the writes of another processor” is through reads, the question becomes, What properties must be enforced among reads and writes to different locations by different processors?

Although the question, how consistent?, seems simple, it is remarkably complicated, as we can see in the following example. Here are two code segments from processes P1 and P2, shown side by side:

```
P1:    A = 0;               P2:    B = 0;
        .....                       .....
       A = 1;                      B = 1;
L1:    if (B == 0) ...      L2:    if (A == 0) ...
```

Assume that the processes are running on different processors, and that locations A and B are originally cached by both processors with the initial value of 0. If writes always take immediate effect and are immediately seen by other processors, it will be impossible for *both* if statements (labeled L1 and L2) to evaluate their conditions as true, since reaching the if statement means that either A or B must have been assigned the value 1. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay; then it is possible that both P1 and P2 have not seen the invalidations for B and A (respectively) *before* they attempt to read the values. The question is, Should this behavior be allowed, and if so, under what conditions?

The most straightforward model for memory consistency is called *sequential consistency.* Sequential consistency requires that the result of any execution be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were interleaved. This eliminates the possibility of some nonobvious execution in the previous example, because the assignments must be completed before the if statements are initiated. Figure 8.38 illustrates why sequential consistency prohibits an execution where both if statements evaluate to true.

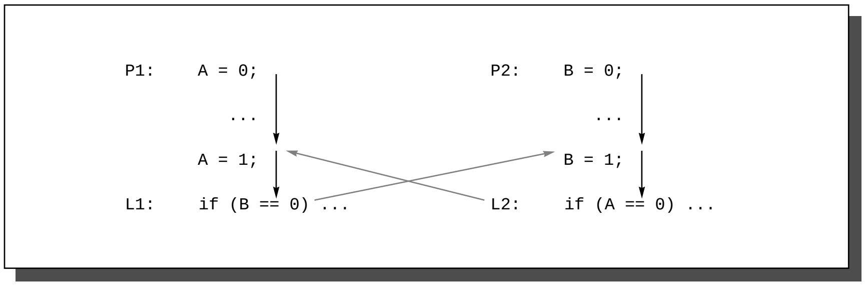


FIGURE 8.38 In sequential consistency, both if statements cannot evaluate to true, since the memory accesses within one process must be kept in program order and the reads of A and B must be interleaved so that one of them completes before the other. To see that this is true, consider the program order shown with black arrows. For both if statements to evaluate to true, the order shown by the two gray arrows must hold, since the reads must appear as if they happen before the writes. For both of these orders to hold and program order to hold, there must be a cycle in the order. The presence of the cycle means that it is impossible to write the accesses down in interleaved order. This means that the execution is not sequentially consistent. You can easily write down all possible orders to help convince yourself.

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally simple to delay the next memory access until the previous one is completed. Remember that memory consistency involves operations among different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of `A` or `B` (`A==0` or `B==0`) until the previous write has completed (`B=1` or `A=1`). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read. Although sequential consistency presents a simple programming paradigm, it reduces potential performance, especially in a machine with a large number of processors, or long interconnect delays, as we can see in the following Example.

EXAMPLE Suppose we have a processor where a write miss takes 40 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 50 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the directory controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

ANSWER When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates. Since the invalidates can overlap, we need only worry about the last one, which starts 10 + 10 + 10 + 10 = 40 cycles after ownership is established. Hence the total time is 40 + 40 + 50 = 130 cycles. In comparison, the ownership time is only 40 cycles. With appropriate write-buffer implementations it is even possible to continue before ownership is established. ■

To provide better performance, designers have developed less restrictive memory consistency models that allow for faster hardware. Such models do affect how the programmer sees the machine, so before we discuss these less restrictive models, let's look at what the programmer expects.

The Programmer's View

Although the sequential consistency model has a performance disadvantage, from the viewpoint of the programmer it has the advantage of simplicity. The challenge is to develop a programming model that is simple to explain and yet allows a high performance implementation. One such programming model that allows us to have a more efficient implementation is to assume that programs are *synchronized*. A program is synchronized if all access to shared data is ordered by synchronization operations. A data reference is ordered by a synchronization operation if, in every possible execution, a write of a variable by one processor and an access (either a read or a write) of that variable by another processor are separated by a pair of synchronization operations, one executed after the write by the writing processor and one executed before the access by the second processor. Cases where variables may be updated without ordering by synchronization are called *data races,* because the execution outcome depends on the relative speed of the processors, and like races in hardware design, the outcome is unpredictable. This leads to another name for synchronized programs: *data-race-free*.

As a simple example, consider a variable being read and updated by two different processors. Each processor surrounds the read and update with a lock and an unlock, both to ensure mutual exclusion for the update and to ensure that the read is consistent. Clearly, every write is now separated from a read by the other processor by a pair of synchronization operations: one unlock (after the write) and one lock (before the read). Of course, if two processors are writing a variable with no intervening reads, then the writes must also be separated by synchronization operations.

We call the synchronization operation corresponding to the unlock a *release,* because it releases a potentially blocked processor, and the synchronization operation corresponding to a lock an *acquire,* because it acquires the right to read the variable. We use the terms acquire and release because they apply to a wide set of synchronization structures, not just locks and unlocks. The next Example shows where the acquires and releases are in several synchronization primitives taken from the previous section.

EXAMPLE Show which operations are acquires and releases in the lock implementation on page 699 and the barrier implementation in Figure 8.34 on page 701.

ANSWER Here is the lock code with the acquire operation shown in bold:

```
lockit:  LL    R2,0(R1)    ;load linked
         BNEZ  R2,lockit   ;not available-spin
         ADDI  R2,R2,#1    ;get locked value
         SC    0(R1),R2    ;store
         BEQZ  R2,lockit   ;branch if store fails
```

The release operation for this lock is simply a store operation (which is not shown, but looks like: sw(R1), R0).

Here is the code for the barrier operation with the acquires shown in bold and the releases in italics (there are two acquires and two releases in the barrier):

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count=count+1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=1;/* release processes */
}
else{ /* more to come */

       spin (release==1);/* wait for arrivals */
}
```

■

We can now define when a program is synchronized using acquires and releases. A program is *synchronized* if every execution sequence containing a write by a processor and a subsequent access of the same data by another processor contains the following sequence of events:

```
write (x)
...
release (s)
...
acquire (s)
...
access(x)
```

It is easy to see that if all such execution sequences look like this, the program is synchronized in the sense that accesses to shared data are always ordered by synchronization and that data races are impossible.

It is a broadly accepted observation that most programs are synchronized. This observation is true primarily because if the accesses were unsynchronized, the behavior of the program would be quite difficult to determine because the speed of execution would determine which processor won a data race and thus affect the results of the program. Even with sequential consistency, reasoning about such programs is very difficult. Programmers could attempt to guarantee ordering by constructing their own synchronization mechanisms, but this is extremely tricky, can lead to buggy programs, and may not be supported architecturally, meaning that they may not work in future generations of the machine. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the machine and the type of synchronization. A standard synchronization library can classify the operations used for synchronization in the library as releases or acquires, or sometimes as both, as, for example, in the case of a barrier.

The major use of unsynchronized accesses is in programs that want to avoid synchronization cost and are willing to accept an inconsistent view of memory. For example, in a stochastic program we may be willing to have a read return an old value of a data item, because the program will still converge on the correct answer. In such cases we still require the system to behave in a coherent fashion, but we do not need to rely on a well-defined consistency model.

Beyond the synchronization operations, we also need to define the ordering of memory operations. There are two types of restrictions on memory orders: *write fences* and *read fences*. Fences are fixed points in a computation that ensure that no read or write is moved across the fence. For example, a write fence executed by processor P ensures that

- all writes by P that occur before P executed the write fence operation have completed, and
- no writes that occur after the fence in P are initiated before the fence.

In sequential consistency, all reads are read fences and all writes are write fences. This limits the ability of the hardware to optimize accesses, since order must be strictly maintained.

From a performance viewpoint, the processor would like to execute reads as early as possible and complete writes as late as possible. Fences act as boundaries, forcing the processor to order reads and writes with respect to the fence. Although a write fence is a two-way blockade, it is most often used to ensure that writes have completed, since the processor wants to delay write completion. Thus the typical effect of a write fence is to cause the program execution to stall until all outstanding writes have completed, including the delivery of any associated invalidations.

A read fence is also a two-way blockade, marking the earliest or latest point that a read may be executed. Most often a read fence is used to mark the earliest point that a read may be executed.

A *memory fence* is an operation that acts as both a read and a write fence. Memory fences enforce ordering among the accesses of different processes. Within a single process we require that program order always be preserved, so reads and writes of the same location cannot be interchanged.

The weaker consistency models discussed in the next section provide the potential for hiding read and write latency by defining fewer read and write fences. In particular, synchronization accesses act as the fences rather than ordinary accesses.

Relaxed Models for Memory Consistency

Since most programs are synchronized and since a sequential consistency model imposes major inefficiencies, we would like to define a more relaxed model that allows higher performance implementations and still preserves a simple programming model for synchronized programs. In fact, there are a number of relaxed models that all maintain the property that the execution semantics of a synchronized program is the same under the model as it would be under a sequential consistency model. The relaxed models vary in how tightly they constrain the set of possible execution sequences, and thus in how many constraints they impose on the implementation.

To understand the variations among the relaxed models and the possible implications for an implementation, it is simplest if we define the models in terms of what orderings among reads and writes *performed by a single processor* are preserved by each model. There are four such orderings:

1. R $\rightarrow$ R: a read followed by a read.
2. R $\rightarrow$ W: a read followed by a write, which is always preserved if the operations are to the same address, since this is an antidependence.
3. W $\rightarrow$ W: a write followed by a write, which is always preserved if they are to the same address, since this is an output dependence.
4. W $\rightarrow$ R: a write followed by a read, which is always preserved if they are to the same address, since this is a true dependence.

If there is a dependence between the read and the write, then uniprocessor program semantics demand that the operations be ordered. If there is no dependence, the memory consistency model determines what orders must be preserved. A sequential consistency model requires that all four orderings be preserved and is thus equivalent to assuming a single centralized memory module that serializes all processor operations, or to assuming that all reads and writes are memory barriers.

When an order is relaxed, it simply means that we allow an operation executed later by the processor to complete first. For example, relaxing the ordering W$\rightarrow$R means that we allow a read that is later than a write to complete before the write has completed. Remember that a write does not complete until all its invalidations complete, so letting the read occur after the write miss has been handled but before the invalidations are done does not preserve the ordering.

A consistency model does not, in reality, restrict the ordering of events. Instead, it says what possible orderings can be *observed*. For example, in sequential consistency, the system must appear to preserve the four orderings just described, although in practice it can allow reordering. This subtlety allows implementations to use tricks that reorder events without allowing the reordering to be observed. Under sequential consistency an implementation can, for example, allow a processor, P, to initiate another write before an earlier write is completed, as long as P does not allow the value of the later write to be seen before the earlier write has completed. For simplicity, we discuss what orderings must be preserved, with the understanding that the implementation has the flexibility to preserve fewer orderings if only the preserved orderings are visible.

The consistency model must also define the orderings imposed between synchronization variable accesses, which act as fences, and all other accesses. When a machine implements sequential consistency, all reads and writes, including synchronization accesses, are fences and are thus kept in order. For weaker models, we need to specify the ordering restrictions imposed by synchronization accesses, as well as the ordering restrictions involving ordinary variables. The simplest ordering restriction is that every synchronization access is a memory fence. If we let S stand for a synchronization variable access, we could also write this with the ordering notation just shown as S→W, S→R, W→S, and R→S. Remember that a synchronization access is also an R or a W and its ordering is affected by other synchronization accesses, which means there is an implied ordering S→S.

The first model we examine relaxes the ordering between a write and a read (to a different address), eliminating the order W→R; this model was first used in the IBM 370 architecture. Such models allow the buffering of writes with bypassing by reads, which occurs whenever the processor allows a read to proceed before it guarantees that an earlier write by that processor has been seen by all the other processors. This model allows a machine to hide some of the latency of a write operation. Furthermore, by relaxing only this one ordering, many applications, even those that are unsynchronized, operate correctly, although a synchronization operation is necessary to ensure that a write completes before a read is done. If a synchronization operation is executed before the read (i.e. a pattern W...S...R), then the orderings W→S and S→R ensure that the write completes before the read. *Processor consistency* and *total store ordering* (TSO) have been used as names for this model, and many machines have implicitly selected this model. This model is equivalent to making the writes be write fences. We summarize all the models, showing the orderings imposed, in Figure 8.39 and show an example in Figure 8.40.

If we also allow nonconflicting writes to potentially complete out of order, by relaxing the W →W ordering, we arrive at a model that has been called *partial store ordering* (PSO). From an implementation viewpoint, it allows pipelining or overlapping of write operations, rather than forcing one operation to complete before another. A write operation need only cause a stall when a synchronization operation, which causes a write fence, is encountered.

The third major class of relaxed models eliminates the R → R and R → W orderings, in addition to the other two orders. This model, which is called *weak ordering*, does not preserve ordering among references, except for the following:

- A read or write is completed before any synchronization operation executed in program order by the processor after the read or write.
- A synchronization operation is always completed before any reads or writes that occur in program order after the operation.

As Figure 8.39 shows, the only orderings imposed in weak order are those created by synchronization operations. Although we have eliminated the $R \rightarrow R$ and $R \rightarrow W$ orderings, the processor can only take advantage of this if it has nonblocking reads. Otherwise the processor implicitly implements these two orders, since no further instructions can be executed until the R is completed. Even with nonblocking reads, the processor may be limited in the advantage it obtains from relaxing the read orderings, since the primary advantage occurs when the R causes a cache miss and the processor is unlikely to be able to keep busy for the tens to hundreds of cycles that handling the cache miss may take. In general, the major advantage of all weaker consistency models comes in hiding write latencies rather than read latencies.

A more relaxed model can be obtained by extending weak ordering. This model, called *release consistency,* distinguishes between synchronization operations that are used to *acquire* access to a shared variable (denoted S_A) and those that *release* an object to allow another processor to acquire access (denoted S_R). Release consistency is based on the observation that in synchronized programs an acquire operation must precede a use of shared data, and a release operation must follow any updates to shared data and also precede the time of the next acquire. This allows us to slightly relax the ordering by observing that a read or write that precedes an acquire need not complete before the acquire, and also that a read or write that follows a release need not wait for the release. Thus the orderings that are preserved involve only S_A and S_R , as shown in Figure 8.39; as the example in Figure 8.40 shows, this model imposes the fewest orders of the five models.

To compare release consistency to weak ordering, consider what orderings would be needed for weak ordering, if we decompose each S in the orderings to S_A and S_R. This would lead to eight orderings involving synchronization accesses and ordinary accesses plus four orderings involving only synchronization accesses. With such a description, we can see that four of the orderings required under weak ordering are *not* imposed under release consistency: $W \rightarrow S_A$, $R\rightarrow S_A$, $S_R \rightarrow R$, and $S_R \rightarrow W$.

Release consistency provides one of the least restrictive models that is easily checkable, and ensures that synchronized programs will see a sequentially consistent execution. While most synchronization operations are either an acquire or a release (an acquire normally reads a synchronization variable and atomically updates it, while a release usually just writes it), some operations, such as a barrier, act as both an acquire and a release and cause the ordering to be equivalent to weak ordering.

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	$R\rightarrow R$, $R\rightarrow W$, $W\rightarrow R$, $W\rightarrow W$	$S\rightarrow W$, $S\rightarrow R$, $R\rightarrow S$, $W\rightarrow S$, $S\rightarrow S$
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	$R\rightarrow R$, $R\rightarrow W$, $W\rightarrow W$	$S\rightarrow W$, $S\rightarrow R$, $R\rightarrow S$, $W\rightarrow S$, $S\rightarrow S$
Partial store order	SPARC	$R\rightarrow R$, $R\rightarrow W$	$S\rightarrow W$, $S\rightarrow R$, $R\rightarrow S$, $W\rightarrow S$, $S\rightarrow S$
Weak ordering	PowerPC		$S\rightarrow W$, $S\rightarrow R$, $R\rightarrow S$, $W\rightarrow S$, $S\rightarrow S$
Release consistency	Alpha, MIPS		$S_A\rightarrow W$, $S_A\rightarrow R$, $R\rightarrow S_R$, $W\rightarrow S_R$, $S_A\rightarrow S_A$, $S_A\rightarrow S_R$, $S_R\rightarrow S_A$, $S_R\rightarrow S_R$

FIGURE 8.39 The orderings imposed by various consistency models are shown for both ordinary accesses and synchronization accesses. The models grow from most restrictive (sequential consistency) to least restrictive (release consistency), allowing increased flexibility in the implementation. The weaker models rely on fences created by synchronization operations, as opposed to an implicit fence at every memory operation. S_A and S_R stand for acquire and release operations, respectively, and are needed to define release consistency. If we used the notation S_A and S_R for each S consistently, each ordering with one S would become two orderings (e.g., $S\rightarrow W$ becomes $S_A\rightarrow W$, $S_R\rightarrow W$), and each $S\rightarrow S$ would become the four orderings shown in the last line of the bottom-right table entry.

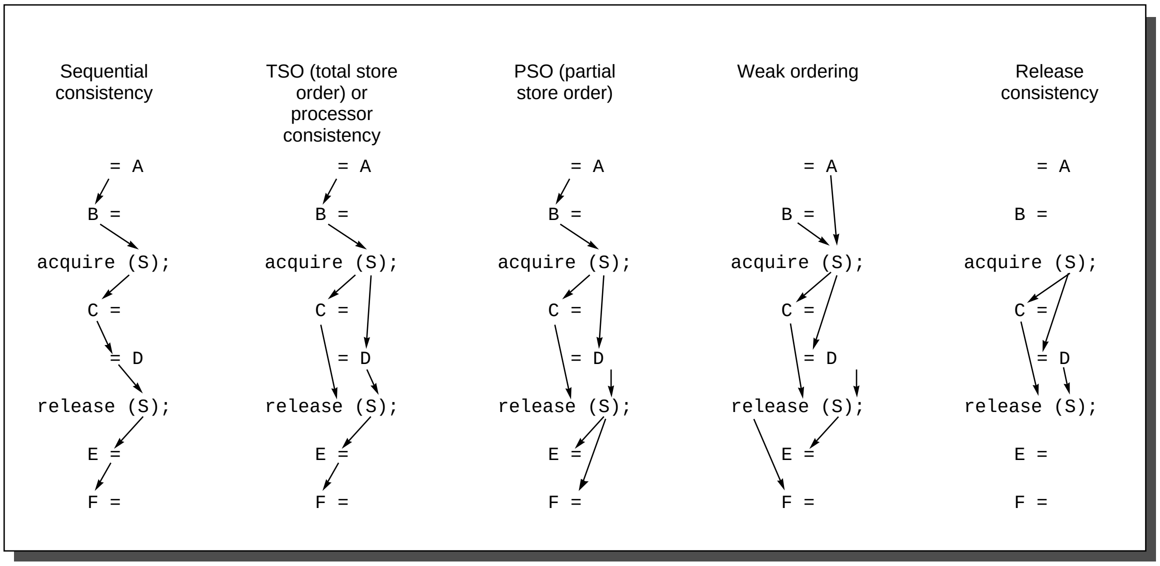


FIGURE 8.40 These examples of the five consistency models discussed in this section show the reduction in the number of orders imposed as the models become more relaxed. Only the minimum orders are shown with arrows. Orders implied by transitivity, such as the write of C before the release of S in the sequential consistency model or the acquire before the release in weak ordering or release consistency, are not shown.

It is also possible to consider even weaker orderings. For example, in release consistency we do not associate memory locations with particular synchronization variables. If we required that the same synchronization variable, V, always be acquired before accessing a particular memory location, M, for example, we could relax the ordering of access to M and acquires and releases of all other synchronization variables other than V. The orderings discussed so far are relatively straightforward to implement. Weaker orderings, such as the previous example, are harder to implement, and it is unclear whether the advantages of weaker orderings would justify their implementation.

Implementation of Relaxed Models

Relaxed models of consistency can usually be implemented with little additional hardware. Most of the complexity lies in implementing memory or interconnect systems that can take advantage of a relaxed model. For example, if the memory or interconnect does not allow multiple outstanding accesses from a processor, then the benefits of the more ambitious relaxed models will be small. Fortunately, most of the benefit can be obtained by having a small number of outstanding writes and one outstanding read.

In this section we describe straightforward implementations of processor consistency and release consistency. Our directory protocols already satisfy the constraints of sequential consistency, since the processor stalls until an operation is complete and the directory first invalidates all sharers before responding to a write miss.

Processor consistency (or TSO) is typically implemented by allowing read misses to bypass pending writes. A write buffer that can support a check to determine whether any pending write in the buffer is to the same address as a read miss, together with a memory and interconnection system that can support two outstanding references per node, is sufficient to implement this scheme. Qualitatively, the advantage of processor consistency over sequential consistency is that it allows the latency of write misses to be hidden.

Release consistency allows additional write latency to be hidden, and if the processor supports nonblocking reads, allows the read latency to be hidden also. To allow write latency to be hidden as much as possible, the processor must allow multiple outstanding writes and allow read misses to bypass outstanding writes. To maximize performance, writes should complete and clear the write buffer as early as possible, which allows any dependent reads to go forward. Supporting early completion of writes requires allowing a write to complete as soon as data are available and before all pending invalidations are completed (since our consistency model allows this). To implement this scheme, either the directory or the original requester can keep track of the invalidation count for each outstanding write. After each invalidation is acknowledged, the pending invalidation count for that write is decreased. We must ensure that all pending invalidates to all outstanding writes complete before we allow a release to complete, so we simply check the pending invalidation counts on any outstanding write when a release is executed. The release is held up until all such invalidations for all outstanding writes complete. In practice, we limit the number of outstanding writes, so that it is easy to track the writes and pending invalidates.

To hide read latency we must have a machine that has nonblocking reads; otherwise, when the processor blocks, little progress will be made. If reads are nonblocking we can simply allow them to execute, knowing that the data dependences will preserve correct execution. It is unlikely, however, that the addition of nonblocking reads to a relaxed consistency model will substantially enhance performance. The limited gain occurs because the read miss times in a multiprocessor are likely to be large and the processor can provide only limited ability to hide this latency. For example, if the reads are nonblocking but the processor executes in order, then the processor will almost certainly block for the read after a few cycles. If the processor supports nonblocking reads and out-of-order execution, it will block as soon as any of its buffers, such as the reorder buffer or reservation stations, are full. (See Chapter 4 for a discussion of full buffer stalls in dynamically scheduled machines.) This is likely to happen in at most tens of cycles, while a miss may cost a hundred cycles. Thus, although the gain may be limited, there is a positive synergy between nonblocking loads and relaxed consistency models.

Performance of Relaxed Models

The performance potential of a more relaxed consistency model depends on both the capabilities of the machine and the particular application. To examine the performance of a memory consistency model, we must first define a hardware environment. The hardware configurations we consider have the following properties:

- The pipeline issues one instruction per clock cycle and is either statically or dynamically scheduled. All functional unit latencies are one cycle.
- Cache misses take 50 clock cycles.
- The CPU includes a write buffer of depth 16.
- The caches are 64 KB and have 16-byte blocks.

To give a flavor of the tradeoffs and performance potential with different hardware capabilities, we consider four hardware models:

1. *SSBR (statically scheduled with blocking reads)*—The processor is statically scheduled and reads that miss in the cache immediately block.

2. *SS (statically scheduled)*—The processor is statically scheduled but reads do not cause the processor to block until the result is used.
3. *DS16 (dynamically scheduled with a 16-entry reorder buffer)*—The processor is dynamically scheduled and has a reorder buffer that allows up to 16 outstanding instructions of any type, including 16 memory access instructions.
4. *DS64 (dynamically scheduled with a 64-entry reorder buffer)*—The processor is dynamically scheduled and has a reorder buffer that allows up to 64 outstanding instructions of any type. This reorder buffer is potentially large enough to hide the total cache miss latency of 50 cycles.

Figure 8.41 shows the relative performance for two of the parallel program benchmarks, LU and Ocean, for these four hardware models and for two different consistency models: total store order (TSO) and release consistency. The performance is shown relative to the performance under a straightforward implementation of sequential consistency. Relaxed models offer a much larger performance gain on Ocean than on LU. This is simply because Ocean has a much higher miss rate and has a significant fraction of write misses. In interpreting the data in Figure 8.41, remember that the caches are fairly small. Most designers would increase the cache size before including nonblocking reads or even beginning to think about dynamic scheduling. This would dramatically reduce the miss rate and the possible advantage from the relaxed model at least for these applications.

Final Remarks on Consistency Models

At the present time, most machines being built support some sort of weak consistency model, varying from processor consistency to release consistency, and almost all also support sequential consistency as an option. Since synchronization is highly machine specific and error prone, the expectation is that most programmers will use standard synchronization libraries and will write synchronized programs, making the choice of a weak consistency model invisible to the programmer and yielding higher performance. Yet to be developed are ideas of how to deal with nondeterministic programs that do not rely on getting the latest values. One possibility is that programmers will not need to rely at all on the timing of updates to variables in such programs; the other possibility is that machine-specific models of update behavior will be needed and used. As remote access latencies continue to increase relative to processor performance, and as features that increase the potential advantage of relaxed models, such as nonblocking caches, are included in more processors, the importance of choosing a consistency model that delivers both a convenient programming model and high performance will increase.

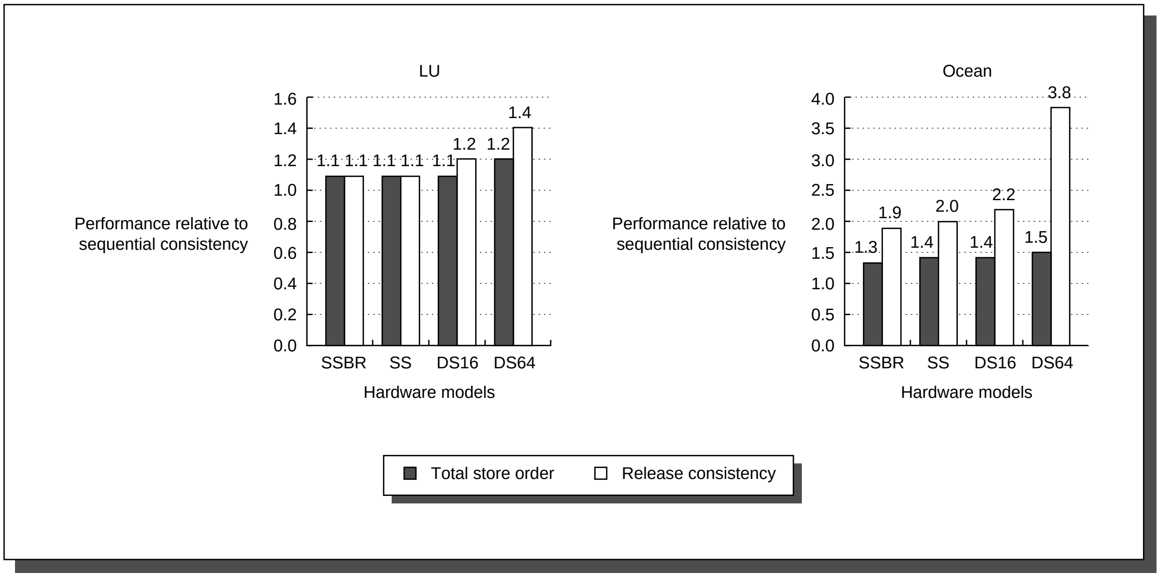


FIGURE 8.41 The performance of relaxed consistency models on a variety of hardware mechanisms, varying from quite reasonable to highly ambitious. The caches are 64 KB, direct mapped, with 16-byte blocks. Misses take 50 cycles. With SSBR most of the write latency is hidden in these benchmarks. It takes dynamic scheduling to hide read latency, and to completely hide read latency a buffer larger than the latency is needed (DS64). For larger cache sizes, the miss rate of Ocean continues to fall and so does the advantage of a relaxed model. For example, at a 256-KB cache with 64-byte blocks and 16 processors, the miss rate is 2%. This leads to an upper bound of 2.0 on the benefits from a relaxed model.

8.7 Crosscutting Issues

Because multiprocessors redefine many system characteristics, they introduce interesting design problems across the spectrum. In this section we give several examples: accurate performance measurement, two examples involving memory systems, an example of the interaction between compilers and the memory consistency model, and a method for using virtual memory support to implement shared memory.

Performance Measurement of Parallel Machines

One of the most controversial issues in parallel processing has been how to measure the performance of parallel machines. Of course, the straightforward answer is to measure a benchmark as supplied and to examine wall-clock time. Measuring wall-clock time obviously makes sense; in a parallel processor, measuring

CPU time can be misleading because the processors may be idle but unavailable for other uses.

Users and designers are often interested in knowing not just how well a machine performs with a certain fixed number of processors, but also how the performance scales as more processors are added. In many cases, it makes sense to scale the application or benchmark, since if the benchmark is unscaled, effects arising from limited parallelism and increases in communication can lead to results that are pessimistic when the expectation is that more processors will be used to solve larger problems. Thus it is often useful to measure the speedup as processors are added both for a fixed-size problem and for a scaled version of the problem, providing an unscaled and a scaled version of the speedup curves. The choice of how to measure the uniprocessor algorithm is also important to avoid anomalous results, since using the parallel version of the benchmark may understate the uniprocessor performance and thus overstate the speedup. This is discussed with an example in section 8.9.

Once we have decided to measure scaled speedup, the question is *how* to scale the application. Let's assume that we have determined that running a benchmark of size n on p processors makes sense. The question is how to scale the benchmark to run on $m \times p$ processors. There are two obvious ways to scale the problem: keeping the amount of memory used per processor constant; and keeping the total execution time, assuming perfect speedup, constant. The first method, called *memory-constrained scaling,* specifies running a problem of size $m \times n$ on $m \times p$ processors. The second method, called *time-constrained scaling,* requires that we know the relationship between the running time and the problem size, since the former is kept constant. For example, suppose the running time of the application with data size n on p processors is proportional to n^2/p. Then with time-constrained scaling, the problem to run is the problem whose ideal running time on $m \times p$ processors is still n^2/p. The problem with this ideal running time has size $\sqrt{m} \times n$.

EXAMPLE Suppose we have a problem whose execution time for a problem of size n is proportional to n^3. Suppose the actual running time on a 10-processor machine is 1 hour. Under the time-constrained and memory-constrained scaling models, find the size of the problem to run and the effective running time for a 100-processor machine.

ANSWER For the time-constrained problem, the ideal running time is the same, 1 hour, so the problem size is $\sqrt[3]{10} \times n$. For memory-constrained scaling, the size of the problem is 10n and the ideal execution time is $10^3/10$, or 100 hours! Since most users will be reluctant to run a problem on an order of magnitude more processors for 100 times longer, this size problem is probably unrealistic. ■

In addition to the scaling methodology, there are questions as to how the program should be scaled when increasing the problem size affects the quality of the result. Since many parallel programs are simulations of physical phenomena, changing the problem size changes the quality of the result, and we must change the application to deal with this effect. As a simple example, consider the effect of time to convergence for solving a differential equation. This time typically increases as the problem size increases, since, for example, we often require more iterations for the larger problem. Thus when we increase the problem size, the total running time may scale faster than the basic algorithmic scaling would indicate. For example, suppose that the number of iterations grows as the log of the problem size. Then for a problem whose algorithmic running time is linear in the size of the problem, the effective running time actually grows proportional to $n \log n$. If we scaled from a problem of size m on 10 processors, purely algorithmic scaling would allow us to run a problem of size 10 m on 100 processors. Accounting for the increase in iterations means that a problem of size $k \times m$, where $k \log k = 10$, will have the same running time on 100 processors. This yields a scaling of 5.72 m, rather than 10 m. In practice, scaling to deal with error requires a good understanding of the application and may involve other factors, such as error tolerances (for example, it affects the cell-opening criteria in Barnes-Hut). In turn, such effects often significantly affect the communication or parallelism properties of the application as well as the choice of problem size.

Scaled speedup is not the same as unscaled (or true) speedup; confusing the two has led to erroneous claims. Scaled speedup has an important role, but only when the scaling methodology is sound and the results are clearly reported as using a scaled version of the application.

Memory System Issues

As we have seen in this chapter, memory system issues are at the core of the design of shared-memory multiprocessors. Indeed, multiprocessing introduces many new memory system complications that do not exist in uniprocessors. In this section we look at two implementation issues that have a significant impact on the design and implementation of a memory system in a multiprocessor context.

Inclusion and Its Implementation

Many multiprocessors use multilevel cache hierarchies to reduce both the demand on the global interconnect and the latency of cache misses. If the cache also provides *multilevel inclusion*—every level of cache hierarchy is a subset of the level further away from the processor—then we can use the multilevel structure to reduce the contention between coherence traffic and processor traffic, as explained earlier. Thus most multiprocessors with multilevel caches enforce the inclusion property. This restriction is also called the *subset property,* because each cache is a subset of the cache below it in the hierarchy.

At first glance, preserving the multilevel inclusion property seems trivial. Consider a two-level example: any miss in L1 either hits in L2 or generates a miss in L2, causing it to be brought into both L1 and L2. Likewise, any invalidate that hits in L2 must be sent to L1, where it will cause the block to be invalidated, if it exists.

The catch is what happens when the block size of L1 and L2 are different. Choosing different block sizes is quite reasonable, since L2 will be much larger and have a much longer latency component in its miss penalty, and thus will want to use a larger block size. What happens to our "automatic" enforcement of inclusion when the block sizes differ? A block in L2 represents multiple blocks in L1, and a miss in L2 causes the replacement of data that is equivalent to multiple L1 blocks. For example, if the block size of L2 is four times that of L1, then a miss in L2 will replace the equivalent of four L1 blocks. Let's consider a detailed example.

EXAMPLE Assume that L2 has a block size four times that of L1. Show how a miss for an address that causes a replacement in L1 and L2 can lead to violation of the inclusion property.

ANSWER Assume that L1 and L2 are direct mapped and that the block size of L1 is b bytes and the block size of L2 is $4b$ bytes. Suppose L1 contains blocks with starting addresses x and $x + b$ and that $x \bmod 4b = 0$, meaning that x also is the starting address of a block in L2. That single block in L2 contains the L1 blocks x, $x + b$, $x + 2b$, and $x + 3b$. Suppose the processor generates a reference to block y that maps to the block containing x in both caches and hence misses. Since L2 missed, it fetches $4b$ bytes and replaces the block containing x, $x + b$, $x + 2b$, and $x + 3b$, while L1 takes b bytes and replaces the block containing x. Since L1 still contains $x + b$, but L2 does not, the inclusion property no longer holds. ■

To maintain inclusion with multiple block sizes, we must probe the higher levels of the hierarchy when a replacement is done at the lower level to ensure that any words replaced in the lower level are invalidated in the higher-level caches. Most systems chose this solution rather than the alternative of not relying on inclusion and snooping the higher-level caches. In the Exercises we explore inclusion further and show that similar problems exist if the associativity of the levels is different.

Nonblocking Caches and Latency Hiding

We saw the idea of nonblocking or lockup-free caches in Chapter 5, where the concept was used to reduce cache misses by overlapping them with execution and by pipelining misses. There are additional benefits in the multiprocessor case. The first is that the miss penalties are likely to be larger, meaning there is more latency to hide, and the opportunity for pipelining misses is also probably larger, since the memory and interconnect system can often handle multiple outstanding memory references.

A machine needs nonblocking caches to take advantage of weak consistency models. For example, to implement a model like processor consistency requires that writes be nonblocking with respect to reads so that a processor can continue either immediately, by buffering the write, or as soon as it establishes ownership of the block and updates the cache. Relaxed consistency models allow further reordering of misses, but nonblocking caches are needed to take full advantage of this flexibility. With the more extensive use of nonblocking caches and dynamic scheduling, we can expect the potential benefits of relaxed consistency models to increase.

Finally, nonblocking support is critical to implementing prefetching. Prefetching, which we also discussed in Chapter 5, is even more important in multiprocessors than in uniprocessors, due to longer memory latencies. In Chapter 5 we described why it is important that prefetches not affect the semantics of the program, since this allows them to be inserted anywhere in the program without changing the results of the computation.

In a multiprocessor, maintaining the absence of any semantic impact from the use of prefetches requires that prefetched data be kept coherent. A prefetched value is kept coherent if, when the value is actually accessed by a load instruction, the most recently written value is returned, even if that value was written after the prefetch. This is exactly the property that cache coherence gives us for other variables in memory. A prefetch that brings a data value closer, and guarantees that on the actual memory access to the data (a load of the prefetched value) the most recent value of the data item is obtained, is called *nonbinding,* since the data value is not bound to a local copy, which would be incoherent. By contrast, a prefetch that moves a data value into a general-purpose register is binding, since the register value is a new variable, as opposed to a cache block, which is a coherent copy of a variable. A nonbinding prefetch maintains the coherence properties of any other value in memory, while a binding prefetch appears more like a register load, since it removes the data from the coherent address space.

Why is nonbinding prefetch critical? Consider a simple but typical example: a data value written by one processor and used by another. In this case, the consumer would like to prefetch the value as early as possible; but suppose the producing process is delayed for some reason. Then the prefetch may fetch the old value of the data item. If the prefetch is nonbinding, the copy of the old data is invalidated when the value is written, maintaining coherence. If the prefetch is binding, however, then the old, incoherent value of the data is used by the prefetching process. Because of the long memory latencies, a prefetch may need to be placed a hundred or more instructions earlier than the data use, if we aim to hide the entire latency. This makes the nonbinding property vital to ensure coherent usage of the prefetch in multiprocessors.

Implementing prefetch requires the same sort of support that a lockup-free cache needs, since there are multiple outstanding memory accesses. This causes several complications:

1. A local node will need to keep track of the multiple outstanding accesses, since the replies may return in a different order than they were sent. This can be handled by adding tags to the requests, or by incorporating the address of the memory block in the reply.
2. Before issuing a request, the node must ensure that it has not already issued a request for the same block, since two requests for the same block could lead to incorrect operation of the protocol. In particular, if the node issues a write miss to a block, while it has such a write miss outstanding both our snooping protocol and directory protocol fail to operate properly.
3. Our implementation of the directory and snooping controllers assumes that the processor stalls on a miss. Stalling allows the cache controller to simply wait for a reply when it has generated a request. With a nonblocking cache, stalling is not possible and the actual implementation must deal with additional processor requests.

Compiler Optimization and the Consistency Model

Another reason for defining a model for memory consistency is to specify the range of legal compiler optimizations that can be performed on shared data. In explicitly parallel programs, unless the synchronization points are clearly defined and the programs are synchronized, the compiler could not interchange a read and a write of two different shared data items, because such transformations might affect the semantics of the program. This prevents even relatively simple optimizations, such as register allocation of shared data, because such a process usually interchanges reads and writes. In implicitly parallelized programs—for example, those written in High Performance FORTRAN (HPF)—programs must be synchronized and the synchronization points are known, so this issue does not arise.

Using Virtual Memory Support to Build Shared Memory

Suppose we wanted to support a shared address space among a group of workstations connected to a network. One approach is to use the virtual memory mechanism and operating system (OS) support to provide shared memory. This approach, which was first explored more than 10 years ago, has been called *distributed virtual memory (DVM)* or *shared virtual memory (SVM).* The key observation that this idea builds on is that the virtual memory hardware has the ability to control access to portions of the address space for both reading and writing. By using the hardware to check and intercept accesses and the operating system to ensure coherence, we can create a coherent, shared address space across the distributed memory of multiple processors.

In SVM, pages become the units of coherence, rather than cache blocks. The OS can allow pages to be replicated in read-only fashion, using the virtual memory support to protect the pages from writing. When a process attempts to write such a page, it traps to the operating system. The operating system on that processor can then send messages to the OS on each node that shares the page, requesting that the page be invalidated. Just as in a directory system, each page has a home node, and the operating system running in that node is responsible for tracking who has copies of the page.

The mechanisms are quite similar to those at work in coherent shared memory. The key differences are that the unit of coherence is a page and that software is used to implement the coherence algorithms. It is exactly these two differences that lead to the major performance differences. A page is considerably bigger than a cache block, and the possibilities for poor usage of a page and for false sharing are very high. This leads to much less stable performance and sometimes even lower performance than a uniprocessor. Because the coherence algorithms are implemented in software, they have much higher overhead.

The result of this combination is that shared virtual memory has become an acceptable substitute for loosely coupled message passing, since in both cases the frequency of communication must be low, and communication that is structured in larger blocks is favored. Distributed virtual memory is not currently competitive with schemes that have hardware-supported, coherent memory, such as the distributed shared-memory schemes we examined in section 8.4: Most programs written for coherent shared memory cannot be run efficiently on shared virtual memory.

Several factors could change the attractiveness of shared virtual memory. Better implementation and small amounts of hardware support could reduce the overhead in the operating system. Compiler technology, as well as the use of smaller or multiple page sizes, could allow the system to reduce the disadvantages of coherence at a page-level granularity. The concept of software-supported shared memory remains an important and active area of research, and such techniques may play an important role in improving the hardware mechanisms; in using more loosely coupled machines, such as networks of workstations; or in allowing coherent shared memory to be extended to larger or more distributed machines possibly built with DSM clusters.

8.8 Putting It All Together: The SGI Challenge Multiprocessor

In this section we examine the design and performance of the Silicon Graphics Challenge multiprocessor. The Challenge is a bus-based, cache-coherent design with a wide, high-speed bus, capable of holding up to 36 MIPS R4400 processors with 4 processors on a board, and up to 16 GB of eight-way interleaved memory. The Power Challenge design uses the much higher performance R8000 (TFP) microprocessors but with the same bus, memory, and I/O system; with two processors per board a Power Challenge can hold up to 18 processors. Our discussion will focus on the bus, coherence hardware, and synchronization support, and our measurements use a 150-MHz R4400-based Challenge system.

The Challenge System Bus (POWERpath-2)

The core of the Challenge design is a wide (256 data bits; 40 address bits), 50-MHz bus (using reduced voltage swing) called POWERpath-2. This bus interconnects all the major system components and provides support for coherence. Figure 8.42 is a diagram of the system configuration.

The POWERpath-2 implements a write-invalidate coherence scheme using four states: the three states we saw in section 8.3 and an additional state representing a block that is clean but exclusive in one cache. The basic coherence protocol is a slight extension over the three-state protocol we examined in detail. When a block that is not shared by any cache is read, the block is placed in the clean exclusive state. Additional read misses of the same block by other processors change the block to the shared state. A write miss causes the block to be transferred to the exclusive (dirty exclusive) state. The major advantage of this protocol is that there is no need to generate a write miss, or an invalidate, when a block is upgraded from clean exclusive to dirty exclusive. Although this is unlikely to be a large effect for accesses to truly shared data (since such data will often be in some processor's cache), it does improve the performance of accesses to private data and to data that are potentially but not actually shared. In particular, data that are not actually shared behave as if the machine were a uniprocessor, where a cache miss would not be needed to write a block already resident in the cache. Implementing the clean exclusive state is straightforward, because the processor sees any attempt to read or write the block and can change the state accordingly.

The POWERpath-2 bus supports split transactions and can have up to eight pending memory reads outstanding, including reads needed to satisfy a write miss. All pending reads are tracked at each processor board, and *resource identifiers,* unique numbers between 0 and 7 (since there are eight outstanding operations), are assigned to reads as they arrive. The resource identifiers are used to tag

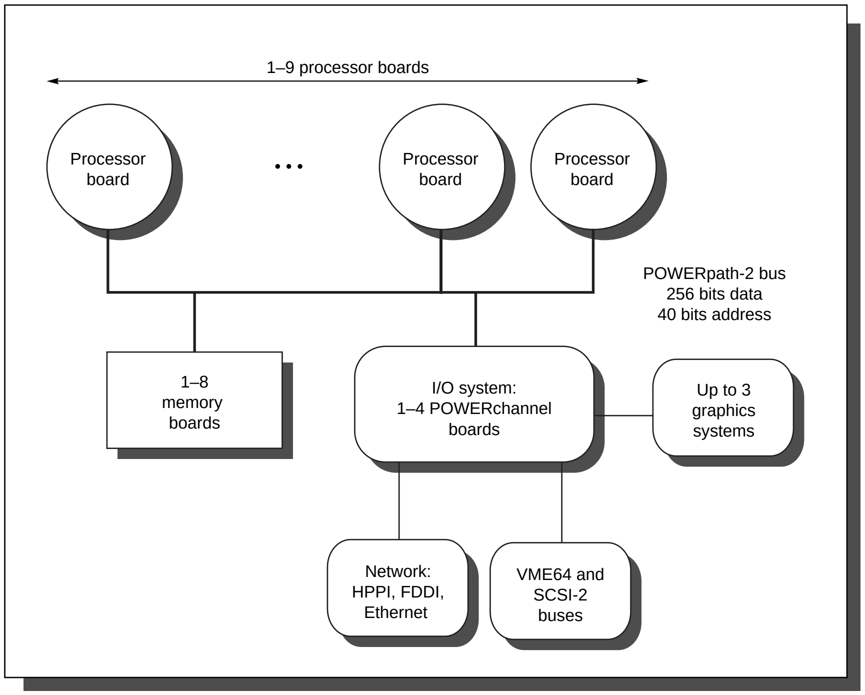


FIGURE 8.42 The Challenge system structure relies on a fast wide bus to interconnect the system and to allow expansion. Each processor board holds four R4400 processors or two R8000 processors. Each memory board holds up to 2 GB. In the largest configuration there are 15 POWERpath-2 bus slots, which can be configured with up to a maximum of nine CPU boards (36 R4400s or 18 R8000s), eight memory boards (16 GB), five VME64 buses (with up to 20 more in an extension box), 32 SCSI-2 channels, eight Ethernets, four HPPI channels, four FDDI connections, and up to three graphics subsystems. This configuration allows four disks in the main cabinet and almost 500 disks with expansion cabinets, for a total of multiple terabytes. For further information about the Challenge design, see http://www.sgi.com/.

results, so that processors know when a response to their request is on the bus. If all eight resource identifiers are in use, a read must wait for one to become free, which happens when an outstanding read completes.

In Appendix E we discuss the difficulties raised by a split transaction bus. In particular, the cache-coherence algorithm of section 8.3 will not work correctly if two writes or a write and a read for the same cache block can be interleaved on the bus. (The interested reader should see Appendix E for an example and further details.) To prevent this problem, a processor board will not issue an invalidate (or write miss) for the same address as any pending read. Thus each processor module keeps track of the requested address for the eight possible outstanding reads. If a write miss occurs for a read that is pending, the processor is stalled and the write is delayed. If a processor board receives a read miss for an address for which a read is pending, the later read can "piggyback" on the earlier read. When the response to the earlier read appears (tagged with the correct resource ID), the bus interface simply grabs the data and uses it to satisfy the later cache miss.

For each of the eight read resources there is an inhibit line, which is used to indicate that a read request should not be responded to by the memory. Snoops can take variable amounts of time, depending on the state of the processor and the block, so this signal is critical to implementing coherence. When a processor receives a snoop request, it asserts the inhibit line until it completes its snoop. If the snoop either finds that the cache does not have the block or finds the block in a clean state, the processor drops the inhibit line. If all the processors drop the inhibit line, the memory will respond, since it knows that all the processor copies are clean. If a processor finds a dirty copy of the block, it requests the bus and places the data on the bus, after which it drops the inhibit line. Both the requesting processor and the memory receive the data and write it. In this latter case, since there are separate tags for snooping, and retrieving a value from the first- or second-level cache takes much longer than detecting the presence of the data, this solution of letting memory respond when its copy is valid is considerably faster than always intervening.

The data and address buses are separately arbitrated and used. Write-back requests use both buses, read responses use only the data bus, and invalidate and read requests use only the address bus. Since the buses are separately arbitrated and assigned, the combined pair of buses can simultaneously carry a new read request and a response to an earlier request.

On POWERpath-2, each bus transaction consists of five bus clock cycles, each 20 ns in length, that are executed in parallel by the bus controllers on each board. These five cycles are arbitration, resolution, address, decode, and acknowledge. On the address bus, one address is sent in the five-cycle bus transaction. On the data bus, four 256-bit data transfers are performed during the five-cycle bus transaction. Thus the sustained bus transfer rate is

$$\begin{aligned}&(256/8) \text{ bytes/transfer} \times (1000/20) \text{ bus cycles/microsecond}\\ &\qquad\times (4/5) \text{ transfers/bus cycle} = 1.28 \text{ GB/sec}\end{aligned}$$

To obtain high bandwidth from memory, each memory board provides a 576-bit path to the DRAMs (512 bits of data and 64 bits of ECC), allowing a single memory cycle to provide the data for two bus transfers. Thus with two-way interleaving a single memory board can support the full bus bandwidth.The total time to satisfy a read miss for a 128-byte cache block with no contention is 22 bus clock cycles:

1. The initial read request is one bus transaction of five bus clock cycles.
2. The latency until memory is ready to transfer is 12 bus clock cycles.

3. The reply transfers all 128 bytes in one reply transaction (four 256-bit transfers), taking five bus clock cycles.

Thus, latency of the access from the processor viewpoint is 22 bus clock cycles, assuming that the memory access can start when the address is received and the reply transaction can start immediately after the data is available. The bus is split transaction, so other requests and replies can occur during the memory access time.

Thus to calculate the secondary cache miss time, we need to compute the latency of each step from the point a memory address is issued until the processor is restarted after the miss:

1. The first step is the initial detection of the miss and generation of the memory request by the processor. This process consists of three steps: detecting a miss in the primary on-chip cache; initiating a secondary (off-chip) cache access and detecting a miss in the secondary cache; and driving the complete address off-chip through the system bus. This process takes about 40 processor clock cycles.
2. The next step is the bus and memory system component, which we know takes 22 bus clock cycles.
3. The next step is reloading the cache line. The R4400 is stalled until the entire cache block is reloaded, so the reload time is added into the miss time. The memory interface on the processor is 64 bits and operates at the external bus timing of 50 MHz, which is the same as the bus timing. Thus reloading the 128-byte cache block takes 16 bus clock cycles.
4. Finally, 10 additional processor clock cycles are used to reload the primary cache and restart the pipeline.

The total miss penalty for a secondary cache miss consists of 50 processor clocks plus 38 bus clocks. For a 150-MHz R4400, each bus clock (20 ns) is three processor clocks (6.67 ns), so the miss time is 164 processor clocks, or 1093 ns. This number is considerably larger than it would be for a uniprocessor memory access, as we discuss in section 8.9. The next section discusses performance.

Performance of the Parallel Program Workload on Challenge

Figure 8.43 shows the speedup for our applications running on up to 16 processors. The speedups for 16 processors vary from 10.5 to 15.0. Because these benchmarks have been tuned to improve memory behavior, the uniprocessor version of the parallel benchmark is better than the original uniprocessor version. Thus we report speedup relative to the uniprocessor parallel version. To understand what's behind the speedups, we can examine the components of the execution time.

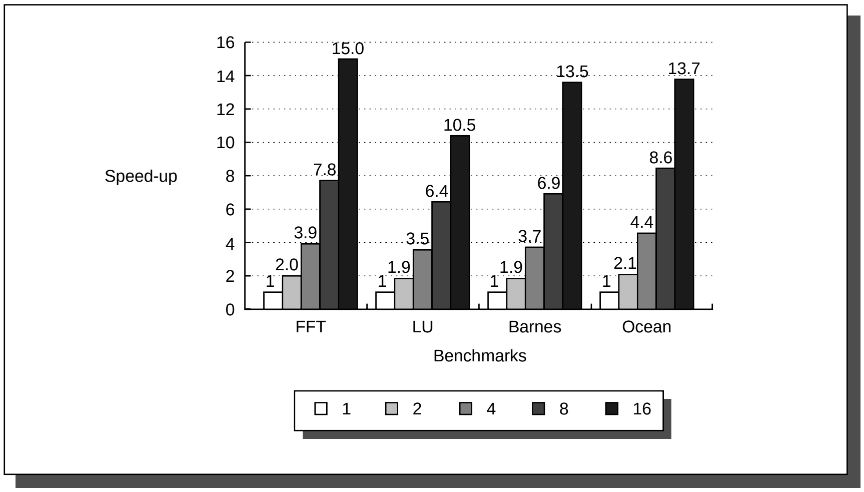


FIGURE 8.43 The speedups for the parallel benchmarks are shown versus processor count for a 150-MHz R4400 Challenge system. The superlinear speedup for Ocean running on two processors occurs because of the doubling of the available cache.

Figure 8.44 shows how the total execution time is composed of three components: CPU time, memory overhead or stall time, and synchronization wait time. With the data in Figure 8.44, we can explain the speedup behavior of these programs. For example, FFT shows the most linear speedup, as can be seen by the nearly constant processor utilization. Both Barnes and Ocean have some drop-off in speedup due to slight increases in synchronization overhead. LU has the most significant loss of speedup, which arises from a significant increase in synchronization overhead. Interestingly, this synchronization overhead actually represents load imbalance, which becomes a serious problem as the ratio between a fixed problem size and the number of processors decreases. Figure 8.44 also reminds us that speedup is a dangerous measure to use by itself. For example, Barnes and Ocean have nearly identical speedup at 16 processors, but the 16 processors are busy 70% of the time in Barnes and only 43% of the time in Ocean. Clearly, Barnes makes much more effective use of the Challenge multiprocessor.

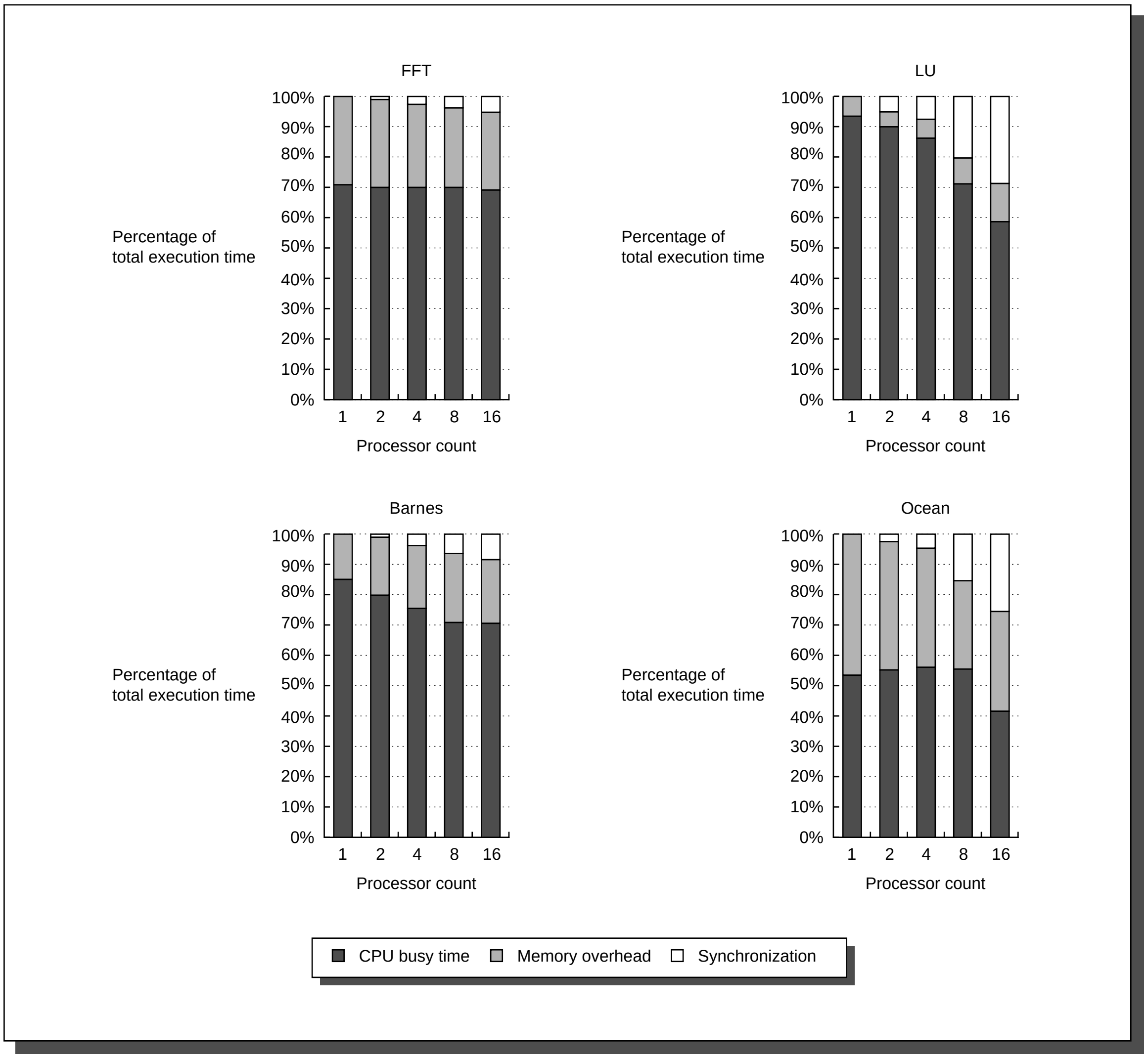


FIGURE 8.44 The components of execution time for the four parallel benchmarks running on a Challenge multiprocessor with 1 to 16 150-MHz R4400. Although memory stalls are a problem with all processor counts, synchronization stalls become more severe as the processor count is increased. Synchronization stalls often result from load balancing, as in LU, or from contention arising when synchronization becomes very frequent, as in Ocean. These applications have been tuned to have good locality and low miss rates, which keep memory stalls from becoming an increasing problem with larger processor counts.

8.9 Fallacies and Pitfalls

Given the lack of maturity in our understanding of parallel computing, there are many hidden pitfalls that will be uncovered either by careful designers or by unfortunate ones. Given the large amount of hype that often surrounds multiprocessors, especially at the high end, common fallacies abound. We have included a selection of these.

Pitfall: Measuring performance of multiprocessors by linear speedup versus execution time.

"Mortar shot" graphs—plotting performance versus number of processors showing linear speedup, a plateau, and then a falling off—have long been used to judge the success of parallel processors. Although speedup is one facet of a parallel program, it is not a direct measure of performance. The first question is the power of the processors being scaled: A program that linearly improves performance to equal 100 Intel 8080s may be slower than the sequential version on a workstation. Be especially careful of floating-point-intensive programs; processing elements without hardware assist may scale wonderfully but have poor collective performance.

Comparing execution times is fair only if you are comparing the best algorithms on each machine. Comparing the identical code on two machines may seem fair, but it is not; the parallel program may be slower on a uniprocessor than a sequential version. Developing a parallel program will sometimes lead to algorithmic improvements, so that comparing the previously best-known sequential program with the parallel code—which seems fair—will not compare equivalent algorithms. To reflect this issue, the terms *relative speedup* (same program) and *true speedup* (best program) are sometimes used. Results that suggest *superlinear* performance, when a program on *n* processors is more than *n* times faster than the equivalent uniprocessor, may indicate that the comparison is unfair, although there are instances where "real" superlinear speedups have been encountered. For example, when Ocean is run on two processors, the combined cache produces a small superlinear speedup (2.1 vs. 2.0), as shown in Figure 8.43.

In summary, comparing performance by comparing speedups is at best tricky and at worst misleading. Comparing the speedups for two different machines does not necessarily tell us anything about the relative performance of the machines. Even comparing two different algorithms on the same machine is tricky, since we must use true speedup, rather than relative speedup, to obtain a valid comparison.

Fallacy: Amdahl's Law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that Amdahl's Law (see section 1.6) had been broken by an MIMD machine. This hardly meant, however, that the law has been overturned for parallel computers; the neglected portion of the program will still limit performance. To understand the basis of the media reports, let's see what Amdahl [1967] originally said:

A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. [p. 483]

One interpretation of the law was that since portions of every program must be sequential, there is a limit to the useful economic number of processors—say 100. By showing linear speedup with 1000 processors, this interpretation of Amdahl's Law was disproved.

The basis for the statement that Amdahl's Law had been "overcome" was the use of scaled speedup.The researchers scaled the benchmark to have a data set size that is 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark. For this particular algorithm the sequential portion of the program was constant independent of the size of the input, and the rest was fully parallel—hence, linear speedup with 1000 processors.

We have already described the dangers of relating scaled speedup as true speedup. Additional problems with this sort of scaling methodology, which can result in unrealistic running times, were examined in section 8.7.

Fallacy: Multiprocessors are "free."

This fallacy has two different interpretations, and both are erroneous. The first is, given that modern microprocessors contain support for snooping caches, we can build small-scale, bus-based multiprocessors for no additional cost in dollars (other than the microprocessor cost) or sacrifice of performance. Many designers believed this to be true and have even tried to build machines to prove it.

To understand why this doesn't work, you need to compare a design with no multiprocessing extensibility against a design that allows for a moderate level of multiprocessing (say 2–4 processors). The 2–4 processor design requires some sort of bus and a coherence controller that is more complicated than the simple memory controller required for the uniprocessor design. Furthermore, the memory access time is almost always faster in the uniprocessor case, since the processor can be directly connected to memory with only a simple single-master bus. Thus the strictly uniprocessor solution typically has better performance and lower cost than the 1-processor configuration of even a very small multiprocessor. For example, a typical Silicon Graphics workstation using 150-MHz R4400 has a miss penalty that is under 80 processor clocks versus the 164 processor clocks seen on a Challenge.

It also became popular in the 1980s to believe that the multiprocessor design was free in the sense that an MP could be quickly constructed from state-of-the-art microprocessors and then quickly updated using newer processors as they became available. This viewpoint ignores the complexity of cache coherence and the challenge of designing high-bandwidth, low-latency memory systems, which for modern processors is extremely difficult. Moreover, there is additional software effort: compilers, operating systems, and debuggers all must be adapted for a parallel system. The next two fallacies are closely related to this one.

Fallacy: Scalability is almost free.

The goal of scalable parallel computing has been a focus of much of the research and a significant segment of the high-end machine development since the mid-1980s. Until recently, it was widely held that you could build scalability into a multiprocessor and then simply offer the machine at any point on the scale from a small number of processors to a large number. The difficulty with this view is that machines that scale to larger processor counts require substantially more investment (in both dollars and design time) in the interprocessor communication network, as well as in aspects such as reliability and reconfigurability.

As an example, consider the CM-5. It provides an interconnection network capable of scaling to 4000 processors, where it can deliver a bisection bandwidth of 20 GB/sec. At a more typical 32- to 64-processor configuration, however, the bisection bandwidth is only 160–320 MB/sec, which is less than what most bus-based systems provide. Furthermore, the cost per CPU is higher than in a bus-based system in this range.

The cost of scalability can be seen even in more limited design ranges, such as very small MP systems (2–8 processors) versus bus-based systems that scale to 16–32 processors. Although a fast 64-bit bus might be adequate for a small machine with fewer than four processors, a larger number of processors requires a wider, higher bandwidth bus (for example, the 256-bit Challenge bus). The user who buys the large system pays for the cost of this high-performance bus. The SPARCCenter 2000 design addresses this by using a multiple bus design with narrower buses. In a small system, only one bus is needed; a larger system can include two buses. (In fact, a version of this design from Cray can have four buses.) Interleaved memory allows transfers to be done simultaneously on both buses. Even in this case, scalability still has a cost, since each processor requires a bus-snooping ASIC for each bus in the system.

Scalability is also not free in software: To build software applications that scale requires significantly more attention to load balance, locality, potential contention for shared resources, and the serial (or partly parallel) portions of the program. Obtaining scalability for real applications, as opposed to toys or small kernels, across factors of more than 10 in processor count, is a *major* challenge. In the future, better compiler technology and performance analysis tools may help with this critical problem.

Pitfall: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.

There is a long history of software lagging behind on massively parallel machines, possibly because the software problems are much harder. Two examples from mainstream, bus-based multiprocessors illustrate the difficulty of developing software for new multiprocessors. The first has to do with not being able to take advantage of a potential architectural capability, and the second arises from the need to optimize the software for a multiprocessor.

The SUN SPARCCenter is a bus-based machine with one or two buses. Memory is distributed on the boards with the processors to create a simple building block consisting of processor, cache, and memory. With this structure, the multiprocessor could also have a fast local access and use the bus only to access remote memory. The SUN operating system, however, was not able to deal with the NUMA (non-uniform memory access) aspect of memory, including such issues as controlling where memory was allocated (local versus global). If memory pages were allocated randomly, then successive runs of the same application could have substantially different performance, and the benefits of fast local access might be small or nonexistent. In addition, providing both a remote and a local access path to memory slightly complicated the design because of timing. Since the software would not have been able to take advantage of faster local memory and the design was believed to be more complicated, the designers decided to require all requests to go over the bus.

Our second example shows the subtle kinds of problems that can arise when software designed for a uniprocessor is adapted to a multiprocessor environment. The SGI operating system protects the page table data structure with a single lock, assuming that page allocation is infrequent. In a uniprocessor this does not represent a performance problem. In a multiprocessor situation, it can become a major performance bottleneck for some programs. Consider a program that uses a large number of pages that are initialized at start-up, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table data structure, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time!).

This page table serialization eliminates parallelism in initialization and has significant impact on overall parallel performance. This performance bottleneck persists even under multiprogramming. For example, suppose we split the parallel program apart into separate processes and run them, one process per processor, so that there is no sharing between the processes. (This is exactly what one user did, since he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the processes—so even the multiprogramming performance is poor. This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminates the problem.

Pitfall: Neglecting data distribution in a distributed shared-memory machine.

Consider the Ocean benchmark running on a 32-processor DSM architecture. As Figure 8.26 (page 687) shows, the miss rate is 3.1%. Because the grid used for the calculation is allocated in a tiled fashion (as described on page 654), 2.5% of the accesses are local capacity misses and 0.6% are remote communication misses needed to access data at the boundary of each grid. Assuming a 30-cycle local memory access cost and a 100-cycle remote memory access cost, the average miss has a cost of 43.5 cycles.

If the grid was allocated in a straightforward fashion by round-robin allocation of the pages, we could expect 1/32 of the misses to be local and the rest to be remote. This would lead to local miss rate of $3.1\% \times 1/32 = 0.1\%$ and a remote miss rate of 3.0%, for an average miss cost of 96.8 cycles. If the average CPI without cache misses is 1.5, and 45% of the instructions are data references, the version with tiled allocation is

$$\frac{1.5 + 45\% \times 3.1\% \times 96.8}{1.5 + 45\% \times 3.1\% \times 43.5} = \frac{1.5 + 1.35}{1.5 + 0.61} = \frac{2.85}{2.11} = 1.35 \text{ times faster}$$

This analysis only considers latency, and assumes that contention effects do not lead to increased latency, which is very optimistic. Round-robin is also not the worst possible data allocation: if the grid fit in a subset of the memory and was allocated to only a subset of the nodes, contention for memory at those nodes could easily lead to a difference in performance of more than a factor of 2.

Fallacy: Linear speedups are needed to make multiprocessors cost-effective.

It is widely recognized that one of the major benefits of parallel computing is to offer a “shorter time to solution” than the fastest uniprocessor. Many people, however, also hold the view that parallel processors cannot be as cost-effective as uniprocessors unless they can achieve perfect linear speedup. This argument says that because the cost of the machine is a linear function of the number of processors, anything less than linear speedup means that the ratio of performance/cost decreases, making a parallel processor less cost-effective than using a uniprocessor.

The problem with this argument is that cost is not only a function of processor count, but also depends on memory (as well as I/O). The effect of including memory in the system cost was pointed out by Wood and Hill [1995], and we use an example from their article to demonstrate the effect of looking at a complete system. They compare a uniprocessor server, the Challenge DM (a deskside unit with one processor and up to 6 GB of memory), against a multiprocessor Challenge XL, the rack-mounted multiprocessor we examined in section 8.8.

(The XL also has faster processors than those of the Challenge DM—150 MHz versus 100 MHz—but we will ignore this difference.)

First, Wood and Hill introduce a cost function: *cost* (p, m), which equals the list price of a machine with p processors and m megabytes of memory. For the Challenge DM:

$$cost(1, m) = \$38{,}400 + \$100 \times m$$

For the Challenge XL:

$$cost(p, m) = \$81{,}600 + \$20{,}000 \times p + \$100 \times m$$

Suppose our computation requires 1 GB of memory on either machine. Then the cost of the DM is $138,400, while the cost of the Challenge XL is $181,600 + $20,000 × p. For different numbers of processors, we can compute what speedups are necessary to make the use of parallel processing on the XL *more* cost effective than that of the uniprocessor. For example, the cost of an 8-processor XL is $341,600, which is about 2.5 times higher than the DM, so if we have a speedup on 8 processors of more than 2.5, the multiprocessor is actually *more* cost effective than the uniprocessor. If we are able to achieve linear speedup, the 8-processor XL system is actually more than *three times* more cost effective! Things get better with more processors: On 16 processors, we need to achieve a speedup of only 3.6, or less than 25% parallel efficiency, to make the multiprocessor as cost effective as the uniprocessor.

The use of a multiprocessor may involve some additional memory overhead, although this number is likely to be small for a shared-memory architecture. If we assume an extremely conservative number of 100% overhead (i.e., double the memory is required on the multiprocessor), the 8-processor machine needs to achieve a speedup of 3.2 to break even, and the 16-processor machine needs to achieve a speedup of 4.3 to break even. Surprisingly, the XL can even be cost effective when compared against a headless workstation used as a server. For example, the cost function for a Challenge S, which can have at most 256 MB of memory, is

$$cost(1, m) = \$16{,}600 + \$100 \times m$$

For problems small enough to fit in 256 MB of memory on both machines, the XL breaks even with a speedup of 6.3 on 8 processors and 10.1 on 16 processors.

In comparing the cost/performance of two computers, we must be sure to include accurate assessments of both total system cost and what performance is achievable. For many applications with larger memory demands, such a comparison can dramatically increase the attractiveness of using a multiprocessor.

8.10 Concluding Remarks

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. …Demonstration is made of the continued validity of the single processor approach. … [p. 483]

Amdahl [1967]

The dream of building computers by simply aggregating processors has been around since the earliest days of computing. However, progress in building and using effective and efficient parallel processors has been slow. This rate of progress has been limited by difficult software problems as well as by a long process of evolving architecture of multiprocessors to enhance usability and improve efficiency. We have discussed many of the software challenges in this chapter, including the difficulty of writing programs that obtain good speedup due to Amdahl's law, dealing with long remote access or communication latencies, and minimizing the impact of synchronization. The wide variety of different architectural approaches and the limited success and short life of many of the architectures to date has compounded the software difficulties. We discuss the history of the development of these machines in section 8.11.

Despite this long and checkered past, progress in the last 10 years leads to some reasons to be optimistic about the future of parallel processing and multiprocessors. This optimism is based on a number of observations about this progress and the long-term technology directions:

1. The use of parallel processing in some domains is beginning to be understood. Probably first among these is the domain of scientific and engineering computation. This application domain has an almost limitless thirst for more computation. It also has many applications that have lots of natural parallelism. Nonetheless, it has not been easy: programming parallel processors even for these applications remains very challenging. Another important, and much larger (in terms of market size), application area is large-scale data base and transaction processing systems. This application domain also has extensive natural parallelism available through parallel processing of independent requests, but its needs for large-scale computation, as opposed to purely access to large-scale storage systems, are less well understood. There are also several contending architectural approaches that may be viable—a point we discuss shortly.

2. It is now widely held that one of the most effective ways to build computers that offer more performance than that achieved with a single-chip micro-

processor is by building a multiprocessor that leverages the significant price/performance advantages of mass-produced microprocessors. This is likely to become more true in the future.

3. Multiprocessors are highly effective for multiprogrammed workloads that are often the dominant use of mainframes and large servers, including file servers, which handle a restricted type of multiprogrammed workload. In the future, such workloads may well constitute a large portion of the market need for higher-performance machines. When the workload wants to share resources, such as file storage, or can efficiently timeshare a resource, such as a large memory, a multiprocessor can be a very efficient host. Furthermore, the OS software needed to efficiently execute multiprogrammed workloads is becoming commonplace.

While there is reason to be optimistic about the growing importance of multiprocessors, many areas of parallel architecture remain unclear. Two particularly important questions are, How will the largest-scale multiprocessors (the massively parallel processors, or MPPs) be built? and What is the role of multiprocessing as a long-term alternative to higher-performance uniprocessors?

The Future of MPP Architecture

Hennessy and Patterson should move MPPs to Chapter 11.

Jim Gray, when asked about coverage of MPPs
in the second edition of this book, alludes to
Chapter 11 bankruptcy protection in U.S. law (1995)

Small-scale multiprocessors built using snooping-bus schemes are extremely cost-effective. Recent microprocessors have even included much of the logic for cache coherence in the processor chip, and several allow the buses of two or more processors to be directly connected—implementing a coherent bus with no additional logic. With modern integration levels, multiple processors can be placed on a board, or even on a single multi-chip module (MCM), resulting in a highly cost-effective multiprocessor. Using DSM technology it is possible to configure such 2–4 processor nodes into a coherent structure with relatively small amounts of additional hardware. It is premature to predict that such architectures will dominate the middle range of processor counts (16–64), but it appears at the present that this approach is the most attractive.

What is totally unclear at the present is how the very largest machines will be constructed. The difficulties that designers face include the relatively small market for very large machines (> 64 nodes and often > $5 million) and the need for machines that scale to larger processor counts to be extremely cost-effective at the lower processor counts where most of the machines will be sold. At the present there appear to be four slightly different alternatives for large-scale machines:

1. Large-scale machines that simply scale up naturally, using proprietary interconnect and communications controller technology. This is the approach that has been followed so far in machines like the Intel Paragon, using a message passing approach, and Cray T3D, using a shared memory without cache coherence. There are two primary difficulties with such designs. First, the machines are not cost-effective at small scales, where the cost of scalability is not valued. Second, these machines have programming models that are incompatible, in varying degrees, with the mainstream of smaller and midrange machines.

2. Large-scale machines constructed from clusters of mid range machines with combinations of proprietary and standard technologies to interconnect such machines. This cluster approach gets its cost-effectiveness through the use of cost-optimized building blocks. In some approaches, the basic architectural model (e.g., coherent shared memory) is extended. The Convex Exemplar fits in this class. The disadvantage of trying to build this extended machine is that more custom design and interconnect are needed. Alternatively, the programming model can be changed from shared memory to message passing or to a different variation on shared memory, such as shared virtual memory, which may be totally transparent. The disadvantage of such designs is the potential change in the programming model; the advantage is that the large-scale machine can make use of more off-the-shelf technology, including standard networks. Another example of such a machine is the SGI Challenge array, which is built from SGI Challenge machines and uses standard HPPI for its interconnect. Overall, this class of machine, while attractive, remains experimental.

3. Designing machines that use off-the-shelf uniprocessor nodes and a custom interconnect. The advantage of such a machine is the cost-effectiveness of the standard uniprocessor node, which is often a repackaged workstation; the disadvantage is that the programming model will probably need to be message passing even at very small node counts. In some application environments where little or no sharing occurs, this may be acceptable. In addition, the cost of the interconnect, because it is custom, can be significant, making the machine costly, especially at small node counts. The IBM SP-2 is the best example of this approach today.

4. Designing a machine using *all* off-the-shelf components, which promises the lowest cost. The leverage in this approach lies in the use of commodity technology everywhere: in the processors (PC or workstation nodes), in the interconnect (high-speed local area network technology, such as ATM), and in the software (standard operating systems and programming languages). Of

course, such machines will use message passing, and communication is likely to have higher latency and lower bandwidth than in the alternative designs. Like the previous class of designs, for applications that do not need high bandwidth or low-latency communication, this approach can be extremely cost-effective. Databases and file servers, for example, may be a good match to these machines. Also, for multiprogrammed workloads, where each user process is independent of the others, this approach is very attractive. Today these machines are built as workstation clusters or as NOWs (networks of workstations) or COWs (clusters of workstations). The VAXCluster approach successfully used this organization for multiprogrammed and transaction-oriented workloads, albeit with minicomputers rather than desktop machines.

Each of these approaches has advantages and disadvantages, and the importance of the shortcomings of any one approach are dependent on the application class. In 1995 it is unclear which if any of these models will win out for larger-scale machines. For some classes of applications, one of these approaches may even become dominant for small to midrange machines. Finally, some hybridization of these ideas may emerge, given the similarity in several of the approaches.

The Future of Microprocessor Architecture

As we saw in Chapter 4, architects are using ever more complex techniques to try to exploit more instruction-level parallelism. As we also saw in that chapter, the prospects for finding ever-increasing amounts of instruction-level parallelism in a manner that is efficient to exploit are somewhat limited. Likewise, there are increasingly difficult problems to be overcome in building memory hierarchies for high-performance processors. Of course, continued technology improvements will allow us to continue to advance clock rate. But the use of technology improvements that allow a faster gate speed alone is not sufficient to maintain the incredible growth of performance that the industry has experienced in the past 10 years. Maintaining a rapid rate of performance growth will depend to an increasing extent on exploiting the dramatic growth in effective silicon area, which will continue to grow much faster than the basic speed of the process technology.

Unfortunately, for the past five or more years, increases in performance have come at the cost of ever-increasing inefficiencies in the use of silicon area, external connections, and power. This diminishing-returns phenomenon has not yet slowed the growth of performance in the mid 1990s, but we cannot sustain the rapid rate of performance improvements without addressing these concerns through new innovations in computer architecture.

Unlike the prophets quoted at the beginning of the chapter, your authors do not believe that we are about to “hit a brick wall” in our attempts to improve single-processor performance. Instead, we may see a gradual slowdown in performance growth, with the eventual growth being limited primarily by improvements in the speed of the technology. When these limitation will become serious is hard to say, but possibly as early as the beginning of the next century. Even if such a slowdown were to occur, performance might well be expected to grow at the annual rate of 1.35 that we saw prior to 1985.

Furthermore, we do not want to rule out the possibility of a breakthrough in uniprocessor design. In the early 1980s, many people predicted the end of growth in uniprocessor performance, only to see the arrival of RISC technology and an unprecedented 10-year growth in performance averaging 1.6 times per year!

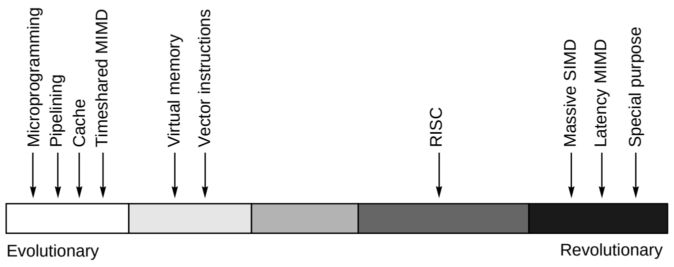
With this in mind, we cautiously ask whether the long-term direction will be to use increased silicon to build multiple processors on a single chip. Such a direction is appealing from the architecture viewpoint—it offers a way to scale performance without increasing complexity. It also offers an approach to easing some of the challenges in memory-system design, since a distributed memory can be used to scale bandwidth while maintaining low latency for local accesses. The challenge lies in software and in what architecture innovations may be used to make the software easier.

Evolution Versus Revolution and the Challenges to Paradigm Shifts in the Computer Industry

Figure 8.45 shows what we mean by the *evolution-revolution spectrum* of computer architecture innovation. To the left are ideas that are invisible to the user (presumably excepting better cost, better performance, or both). This is the evolutionary end of the spectrum. At the other end are revolutionary architecture ideas. These are the ideas that require new applications from programmers who must learn new programming languages and models of computation, and must invent new data structures and algorithms.

Revolutionary ideas are easier to get excited about than evolutionary ideas, but to be adopted they must have a much higher payoff. Caches are an example of an evolutionary improvement. Within 5 years after the first publication about caches, almost every computer company was designing a machine with a cache. The RISC ideas were nearer to the middle of the spectrum, for it took closer to 10 years for most companies to have a RISC product. Most multiprocessors have tended to the revolutionary end of the spectrum, with the largest-scale machines (MPPs) being more revolutionary than others. Most programs written to use multiprocessors as parallel engines have been written especially for that class of machines, if not for the specific architecture.

The challenge for both hardware and software designers that would propose that multiprocessors and parallel processing become the norm, rather than the exception, is the disruption to the established base of programs. There are two possible ways this paradigm shift could be facilitated: if parallel processing offers the only alternative to enhance performance, and if advances in hardware and software technology can construct a gentle ramp that allows the movement to parallel processing, at least with small numbers of processors, to be more evolutionary.



User compatibility	**Binary**	**Upward binary**	**Assembly**	**High-level language**	**New programs, extended or new HLL, new algorithms**
Example	VAX-11/780 vs. 8800	Intel 8086 vs. 80286 vs. 80386 vs. 80486	MIPS 1000 vs. DEC 3100	Sun 3 vs. Sun 4	SISD vs. Intel Paragon
Difference	Microcode, TLB, caches, pipelining, MIMD	Some new instructions	Byte order (Big vs. Little Endian)	Full instruction set (same data representation)	Algorithms, extended HLL, programs

FIGURE 8.45 The evolution-revolution spectrum of computer architecture. The second through fifth columns are distinguished from the final column in that applications and operating systems can be ported from other computers rather than written from scratch. For example, RISC is listed in the middle of the spectrum because user compatibility is only at the level of high-level languages, while microprogramming allows binary compatibility, and latency-oriented MIMDs require changes to algorithms and extending HLLs. Timeshared MIMD means MIMDs justified by running many independent programs at once, while latency MIMD means MIMDs intended to run a single program faster.

8.11 Historical Perspective and References

There is a tremendous amount of history in parallel processing; in this section we divide our discussion by both time period and architecture. We start with the SIMD approach and the Illiac IV. We then turn to a short discussion of some other early experimental machines and progress to a discussion of some of the great debates in parallel processing. Next we discuss the historical roots of the present machines and conclude by discussing recent advances.

The Rise and Fall of SIMD Computers

The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components.... Centralizing the [control unit] gives rise to the basic organization of [an]... array processor such as the Illiac IV.

Bouknight et al. [1972]

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. The key idea in that machine, as in more recent SIMD machines, is to have a single instruction that operates on many data items at once, using many functional units.

The earliest ideas on SIMD-style computers are from Unger [1958] and Slotnick, Borck, and McReynolds [1962]. Slotnick's Solomon design formed the basis of the Illiac IV, perhaps the most infamous of the supercomputer projects. While successful in pushing several technologies that proved useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned machine. Actual performance was at best 15 MFLOPS, versus initial predictions of 1000 MFLOPS for the full system [Hord 1982]. Delivered to NASA Ames Research in 1972, the computer took three more years of engineering before it was usable. These events slowed investigation of SIMD, with Danny Hillis [1985] resuscitating this style in the Connection Machine, which had 65,636 1-bit processors.

Real SIMD computers need to have a mixture of SISD and SIMD instructions. There is an SISD host computer to perform operations such as branches and address calculations that do not need parallel operation. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. For flexibility, individual execution units can be disabled during a SIMD instruction. In addition, massively parallel SIMD machines rely on interconnection or communication networks to exchange data between processing elements.

SIMD works best in dealing with arrays in for-loops. Hence, to have the opportunity for massive parallelism in SIMD there must be massive amounts of data, or *data parallelism*. SIMD is at its weakest in case statements, where each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

The basic trade-off in SIMD machines is performance of a processor versus number of processors. Recent machines emphasize a large degree of parallelism over performance of the individual processors. The Connection Machine 2, for example, offered 65,536 single bit-wide processors, while the Illiac IV had 64 64-bit processors.

After being resurrected in the 1980s, first by Thinking Machines and then by MasPar, the SIMD model has once again been put to bed as a general-purpose multiprocessor architecture, for two main reasons. First, it is too inflexible. A number of important problems cannot use such a style of machine, and the architecture does not scale down in a competitive fashion; that is, small-scale SIMD machines often have worse cost/performance compared with that of the alternatives. Second, SIMD cannot take advantage of the tremendous performance and cost advantages of microprocessor technology. Instead of leveraging this low-cost technology, designers of SIMD machines must build custom processors for their machines.

Although SIMD computers have departed from the scene as general-purpose alternatives, this style of architecture will continue to have a role in special-purpose designs. Many special-purpose tasks are highly data parallel and require a limited set of functional units. Thus designers can build in support for certain operations, as well as hardwire interconnection paths among functional units. Such organizations are often called *array processors*, and they are useful for tasks like image and signal processing.

Other Early Experiments

It is difficult to distinguish the first multiprocessor. Surprisingly, the first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve availability. Holland [1959] gave early arguments for multiple processors.

Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie Mellon University. The first of these was C.mmp [Wulf and Bell 1972; Wulf and Harbison 1978], which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared-memory programming model. Much of the focus of the research in the C.mmp project was on software, especially in the OS area. A later machine, Cm* [Swan et al. 1977], was a cluster-based multiprocessor with a distributed memory and a nonuniform access time. The absence of caches and a long remote access latency made data placement critical. This machine and a number of application experiments are well described by Gehringer, Siewiorek, and Segall [1987]. Many of the ideas in these machines would be reused in the 1980s when the microprocessor made it much cheaper to build multiprocessors.

Great Debates in Parallel Processing

The quotes at the beginning of this chapter give the classic arguments for abandoning the current form of computing, and Amdahl [1967] gave the classic reply in support of continued focus on the IBM 370 architecture. Arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea 1842]! Yet the effectiveness of the multiprocessor for reducing latency of individual important programs is still being explored. Aside from these debates about the advantages and limitations of parallelism, several hot debates have focused on how to build multiprocessors.

How to Build High-Performance Parallel Processors

One of the longest-raging debates in parallel processing has been over how to build the fastest multiprocessors—using many small processors or a smaller number of faster processors. This debate goes back to the 1960s and 1970s. Figure 8.46 shows the state of the industry in 1990, plotting number of processors

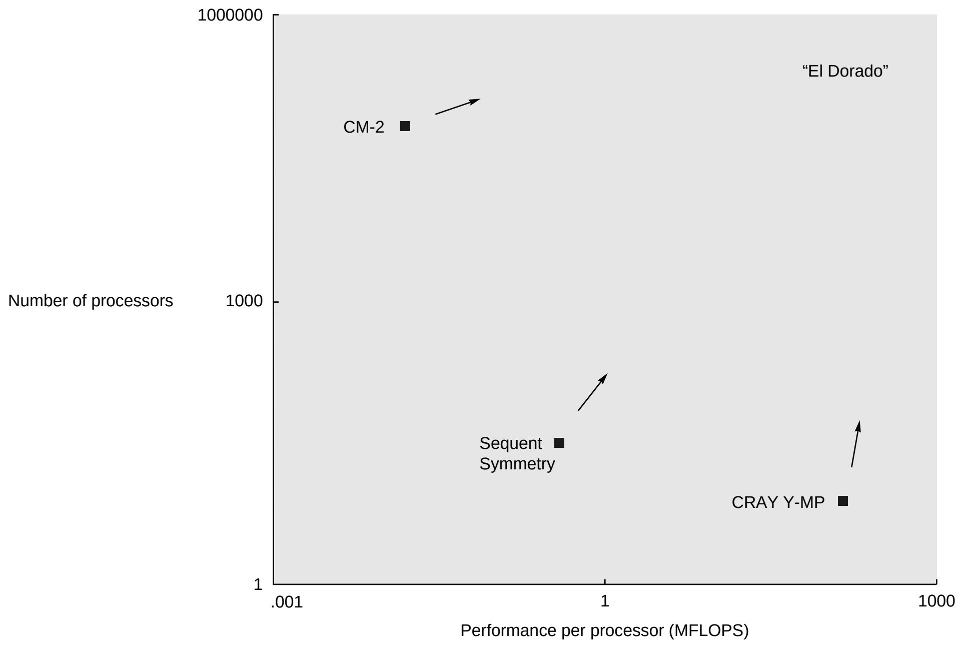


FIGURE 8.46 Danny Hillis, architect of the Connection Machines, has used a figure similar to this to illustrate the multiprocessor industry. (Hillis's x-axis was processor width rather than processor performance.) Processor performance on this graph is approximated by the MFLOPS rating of a single processor for the DAXPY procedure of the Linpack benchmark for a 1000 x 1000 matrix. Generally it is easier for programmers when moving to the right, while moving up is easier for the hardware designer because there is more hardware replication. The massive parallelism question is, Which is the quickest path to the upper right corner? The computer design question is, Which has the best cost/performance or is more scalable for equivalent cost/performance?

versus performance of an individual processor. The massive parallelism question is whether taking the high road or the low road in Figure 8.46 will get us to El Dorado—the highest-performance multiprocessor. In the last few years, much of this debate has subsided. Microprocessor-based machines are assumed to be the basis for the highest-performance multiprocessors. Perhaps the biggest change is the perception that machines built from microprocessors will probably have hundreds and perhaps a few thousand processors, but not the tens of thousands that had been predicted earlier.

In the last five years, the middle road has emerged as the most viable direction. It combines moderate numbers of high-performance microprocessors. This road relies on advances in our ability to program parallel machines as well as on continued progress in microprocessor performance and advances in parallel architecture.

Predictions of the Future

It's hard to predict the future, yet in 1989 Gordon Bell made two predictions for 1995. We included these predictions in the first edition of the book, when the outcome was completely unclear. We discuss them in this section, together with an assessment of the accuracy of the prediction.

The first is that a computer capable of sustaining a teraFLOPS—one million MFLOPS—will be constructed by 1995, either using a multicomputer with 4K to 32K nodes or a Connection Machine with several million processing elements [Bell 1989]. To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1989 the winner used an eight-processor Cray Y-MP to run at 1680 MFLOPS. On the basis of these numbers, machines and programs would have to have improved by a factor of 3.6 each year for the fastest program to achieve 1 TFLOPS in 1995. In 1994, the winner achieved 140,000 MFLOPS (0.14 TFLOPS) using a 1904-node Paragon, which contains 3808 processors. This represents a year-to-year improvement of 2.4, which is still quite impressive.

What has become recognized since 1989 is that although we may have the technology to build a teraFLOPS machine, it is not clear either that anyone could afford it or that it would be cost-effective. For example, based on the 1994 winner, a sustained teraFLOPS would require a machine that is about seven times larger and would likely cost close to $100 million. If factors of 2 in year-to-year performance improvement can be sustained, the price of a teraFLOPS might reach a reasonable level in 1997 or 1998. Gordon Bell argued this point in a series of articles and lectures in 1992–93, using the motto "No teraFLOPS before its time."

The second Bell prediction concerns the number of data streams in supercomputers shipped in 1995. Danny Hillis believed that although supercomputers with a small number of data streams may be the best sellers, the biggest machines will be machines with many data streams, and these will perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995 more sustained MFLOPS will be shipped in machines using few data streams (≤100) rather than many data streams (≥1000). This bet concerns only supercomputers, defined as machines costing more than $1 million and used for scientific applications. Sustained MFLOPS is defined for this bet as the number of floating-point operations per *month*, so availability of machines affects their rating. The loser must write and publish an article explaining why his prediction failed; your authors will act as judge and jury.

In 1989, when this bet was made, it was totally unclear who would win. Although it is a little too early to convene the court, a survey of the current publicly known supercomputers shows only six machines in existence in the world with more than 1000 data streams. It is quite possible that during the last quarter of 1995, *no* machines with ≥1000 data streams will ship. In fact, it appears that much smaller microprocessor-based machines (≤ 20 programs) are becoming dominant. A recent survey of the 500 highest-performance machines in use (based on Linpack ratings), called the Top 500, showed that the largest number of machines were bus-based shared-memory multiprocessors!

More Recent Advances and Developments

With the exception of the parallel vector machines (see Appendix B), all other recent MIMD computers have been built from off-the-shelf microprocessors using a bus and logically central memory or an interconnection network and a distributed memory. A number of experimental machines built in the 1980s further refined and enhanced the concepts that form the basis for many of today's multiprocessors.

The Development of Bus-Based Coherent Machines

Although very large mainframes were built with multiple processors in the 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggests the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware, and that portable operating systems meant that multiprocessor projects no longer required the invention of a new operating system. In this paper, Bell defines the terms *multiprocessor* and *multicomputer* and sets the stage for two different approaches to building larger-scale machines.

The first bus-based multiprocessor with snooping caches was the Synapse N+1 described by Frank [1984]. Goodman [1983] wrote one of the first papers to describe snooping caches. The late 1980s saw the introduction of many commercial bus-based, snooping-cache architectures, including the Silicon Graphics 4D/240 [Baskett et al. 1988], the Encore Multimax [Wilson 1987], and the Sequent Symmetry [Lovett and Thakkar 1988]. The mid 1980s saw an explosion in the development of alternative coherence protocols, and Archibald and Baer [1986] provide a good survey and analysis, as well as references to the original papers.

Toward Large-Scale Multiprocessors

In the effort to build large-scale multiprocessors, two different directions were explored: message passing multicomputers and scalable shared-memory multiprocessors. Although there had been many attempts to build mesh and hypercube-connected multiprocessors, one of the first machines to successfully bring together all the pieces was the Cosmic Cube built at Caltech [Seitz 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. The Intel iPSC 860, a hypercube-connected collection of i860s, was based on these ideas. More recent machines, such as the Intel Paragon, have used networks with lower dimensionality and higher individual links. The Paragon also employed a separate i860 as a communications controller in each node, although a number of users have found it better to use both i860 processors for computation as well as communication. The Thinking Machines CM-5 made use of off-the-shelf microprocessors and a fat tree interconnect (see Chapter 7). It provided user-level access to the communication channel, thus significantly improving communication latency. In 1995, these two machines represent the state of the art in message-passing multicomputers.

Early attempts at building a scalable shared-memory multiprocessor include the IBM RP3 [Pfister et al. 1985], the NYU Ultracomputer [Schwartz 1980; Elder et al. 1985], the University of Illinois Cedar project [Gajksi et al. 1983], and the BBN Butterfly and Monarch [BBN Laboratories 1986; Rettberg et al. 1990]. These machines all provided variations on a nonuniform distributed-memory model, but did not support cache coherence, which substantially complicated programming. The RP3 and Ultracomputer projects both explored new ideas in synchronization (fetch-and-operate) as well as the idea of combining references in the network. In all four machines, the interconnect networks turned out to be more costly than the processing nodes, raising problems for smaller versions of the machine. The Cray T3D builds on these ideas, using a noncoherent shared address space but building on the advances in interconnect technology developed in the multicomputer domain.

Extending the shared-memory model with scalable cache coherence was done by combining a number of ideas. Directory-based techniques for cache coherence were actually known before snooping cache techniques. In fact, the first cache-coherence protocols actually used directories, as described by Tang [1976] and implemented in the IBM 3081. Censier and Feautrier [1978] described a directory coherence scheme with tags in memory. The idea of distributing directories with the memories to obtain a scalable implementation of cache coherence (now called distributed shared memory, or DSM) was first described by Agarwal et al. [1988] and served as the basis for the Stanford DASH multiprocessor (see Lenoski et al. [1990, 1992]). The Kendall Square Research KSR-1 [Burkhardt et al.1992] was the first commercial implementation of scalable coherent shared memory. It extended the basic DSM approach to implement a concept called *COMA* (*cache-only memory architecture*), which makes the main memory a cache, as described in Exercise 8.13. The Convex Exemplar implements scalable coherent shared memory using a two-level architecture: at the lowest level eight-processor modules are built using a crossbar. A ring can then connect up to 32 of these modules, for a total of 256 processors.

Developments in Synchronization and Consistency Models

A wide variety of synchronization primitives have been proposed for shared-memory multiprocessors. Mellor-Crummey and Scott [1991] provide an overview of the issues as well as efficient implementations of important primitives, such as locks and barriers. An extensive bibliography supplies references to other important contributions, including developments in spin locks, queuing locks, and barriers.

Lamport [1979] introduced the concept of sequential consistency and what correct execution of parallel programs means. Dubois, Scheurich, and Briggs [1988] introduced the idea of weak ordering (originally in 1986). In 1990, Adve and Hill provided a better definition of weak ordering and also defined the concept of data-race-free; at the same conference, Gharachorloo [1990] and his colleagues introduced release consistency and provided the first data on the performance of relaxed consistency models.

Other References

There is an almost unbounded amount of information on multiprocessors and multicomputers: Conferences, journal papers, and even books seem to appear faster than any single person can absorb the ideas. No doubt many of these papers will go unnoticed—not unlike the past. Most of the major architecture conferences contain papers on multiprocessors. An annual conference, *Supercomputing XY* (where X and Y are the last two digits of the year), brings together users, architects, software developers, and vendors and publishes the proceedings in book and CD-ROM form. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems,* contain papers on all aspects of parallel processing. Several books focusing on parallel processing are included in the following references. Eugene Miya of NASA Ames has collected an online bibliography of parallel-processing papers that contains more than 10,000 entries. To get information about receiving the bibliography, see http://unix.hensa.ac.uk/parallel/bibliographies/parallelism-biblos-FAQ. Also see [Miya 1985]. In addition to documenting the discovery of concepts now used in practice, these references also provide descriptions of many ideas that have been explored and found wanting, as well as ideas whose time has just not yet come.

References

ADVE, S. V. AND M. D. HILL [1990]. “Weak ordering—A new definition,” *Proc. 17th Int’l Symposium on Computer Architecture* (June), Seattle, 2–14.

AGARWAL, A., J. L. HENNESSY, R. SIMONI, AND M.A. HOROWITZ [1988]. “An evaluation of directory schemes for cache coherence,” *Proc. 15th Int’l Symposium on Computer Architecture* (June), 280–289.

ALMASI, G. S. AND A. GOTTLIEB [1989]. *Highly Parallel Computing,* Benjamin/Cummings, Redwood City, Calif.

AMDAHL, G. M. [1967]. “Validity of the single processor approach to achieving large scale computing capabilities,” *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N.J. (April), 483–485.

ARCHIBALD, J. AND J.-L. BAER [1986]. “Cache coherence protocols: Evaluation using a multiprocessor simulation model,” *ACM Trans. on Computer Systems* 4:4 (November), 273–298.

BASKETT, F., T. JERMOLUK, AND D. SOLOMON [1988]. “The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 10,000 lighted polygons per second,” *Proc. COMPCON Spring,* San Francisco, 468–471.

BBN LABORATORIES [1986]. “Butterfly parallel processor overview,” Tech. Rep. 6148, BBN Laboratories, Cambridge, Mass.

BELL, C. G. [1985]. “Multis: A new class of multiprocessor computers,” *Science* 228 (April 26), 462–467.

BELL, C. G. [1989]. “The future of high performance computers in science and engineering,” *Comm. ACM* 32:9 (September), 1091–1101.

BOUKNIGHT, W. J, S. A. DENEBERG, D. E. MCINTYRE, J. M. RANDALL, A. H. SAMEH, AND D. L. SLOTNICK [1972]. “The Illiac IV system,” *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples,* McGraw-Hill, New York (1982), 306–316.

BURKHARDT, H. III, S. FRANK, B. KNOBE, AND J. ROTHNIE [1992]. “Overview of the KSR1 computer system,” Tech. Rep. KSR-TR-9202001, Kendall Square Research, Boston (February).

CENSIER, L. AND P. FEAUTRIER [1978]. “A new solution to coherence problems in multicache systems,” *IEEE Trans. on Computers* C-27:12 (December), 1112–1118.

DUBOIS, M., C. SCHEURICH, AND F. BRIGGS [1988]. “Synchronization, coherence, and event ordering,” *IEEE Computer* 9-21 (February).

EGGERS, S. [1989]. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors,* Ph.D. Thesis, Univ. of California, Berkeley. Computer Science Division Tech. Rep. UCB/CSD 89/501 (April).

ELDER, J., A. GOTTLIEB, C. K. KRUSKAL, K. P. MCAULIFFE, L. RANDOLPH, M. SNIR, P. TELLER, AND J. WILSON [1985]. “Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach,” *Proc. 12th Int’l Symposium on Computer Architecture* (June), Boston, 126–135.

FLYNN, M. J. [1966]. “Very high-speed computing systems,” *Proc. IEEE* 54:12 (December), 1901–1909.

FRANK, S. J. [1984] “Tightly coupled multiprocessor systems speed memory access time,” *Electronics* 57:1 (January), 164–169.

GAJSKI, D., D. KUCK, D. LAWRIE, AND A. SAMEH [1983]. “CEDAR—A large scale multiprocessor,” *Proc. Int’l Conf. on Parallel Processing* (August), 524–529.

GEHRINGER, E. F., D. P. SIEWIOREK, AND Z. SEGALL [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass.

GHARACHORLOO, K., D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA, AND J. L. HENNESSY [1990]. “Memory consistency and event ordering in scalable shared-memory multiprocessors,” *Proc. 17th Int’l Symposium on Computer Architecture* (June), Seattle, 15–26.

GOODMAN, J. R. [1983]. “Using cache memory to reduce processor memory traffic,” *Proc. 10th Int’l Symposium on Computer Architecture* (June), Stockholm, Sweden, 124–131.

HILLIS, W. D. [1985]. *The Connection Machine,* MIT Press, Cambridge, Mass.

HOCKNEY, R. W. AND C. R. JESSHOPE [1988]. *Parallel Computers-2, Architectures, Programming and Algorithms,* Adam Hilger Ltd., Bristol, England.

HOLLAND, J. H. [1959]. “A universal computer capable of executing an arbitrary number of subprograms simultaneously,” *Proc. East Joint Computer Conf.* 16, 108–113.

HORD, R. M. [1982]. *The Illiac-IV, The First Supercomputer,* Computer Science Press, Rockville, Md.

HWANG, K. [1993]. *Advanced Computer Architecture and Parallel Programming,* McGraw-Hill, New York.

LAMPORT, L. [1979]. “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. on Computers* C-28:9 (September), 241–248.

LENOSKI, D., J. LAUDON, K. GHARACHORLOO, A. GUPTA, AND J. L. HENNESSY [1990]. “The Stanford DASH multiprocessor,” *Proc. 17th Int’l Symposium on Computer Architecture* (June), Seattle, 148–159.

LENOSKI, D., J. LAUDON, K. GHARACHORLOO, W.-D. WEBER, A. GUPTA, J. L. HENNESSY, M. A. HOROWITZ, AND M. LAM [1992]. “The Stanford DASH multiprocessor,” *IEEE Computer* 25:3 (March).

LOVETT, T. AND S. THAKKAR [1988]. “The Symmetry multiprocessor system,” *Proc. 1988 Int’l Conf. of Parallel Processing,* University Park, Penn., 303–310.

MELLOR-CRUMMEY, J. M. AND M. L. SCOTT [1991]. “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Trans. on Computer Systems* 9:1 (February), 21–65.

MENABREA, L. F. [1842]. “Sketch of the analytical engine invented by Charles Babbage,” Bibiothèque Universelle de Genève (October).

MITCHELL, D. [1989]. “The Transputer: The time is now,” *Computer Design* (RISC supplement), 40–41.

MIYA, E. N. [1985]. “Multiprocessor/distributed processing bibliography,” *Computer Architecture News* (ACM SIGARCH) 13:1, 27–29.

PFISTER, G. F., W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFEKDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON, AND J. WEISS [1985]. “The IBM research parallel processor prototype (RP3): Introduction and architecture,” *Proc. 12th Int’l Symposium on Computer Architecture* (June), Boston, 764–771.

RETTBERG, R. D., W. R. CROWTHER, P. P. CARVEY, AND R. S. TOWLINSON [1990]. “The Monarch parallel processor hardware design,” *IEEE Computer* 23:4 (April).

ROSENBLUM, M., S. A. HERROD, E. WITCHEL, AND A. GUTPA [1995]. “Complete computer simulation: The SimOS approach,” to appear in *IEEE Parallel and Distributed Technology* 3:4 (fall).

SCHWARTZ, J. T. [1980]. “Ultracomputers,” *ACM Trans. on Programming Languages and Systems* 4:2, 484–521.

SEITZ, C. [1985]. “The Cosmic Cube,” *Comm. ACM* 28:1 (January), 22–31.

SLOTNICK, D. L., W. C. BORCK, AND R. C. MCREYNOLDS [1962]. “The Solomon computer,” *Proc. Fall Joint Computer Conf.* (December), Philadelphia, 97–107.

STONE, H. [1991]. *High Performance Computers,* Addison-Wesley, New York.

SWAN, R. J., A. BECHTOLSHEIM, K. W. LAI, AND J. K. OUSTERHOUT [1977]. “The implementation of the Cm* multi-microprocessor,” *Proc. AFIPS National Computing Conf.,* 645–654.

SWAN, R. J., S. H. FULLER, AND D. P. SIEWIOREK [1977]. “Cm*—A modular, multi-microprocessor,” *Proc. AFIPS National Computer Conf.* 46, 637–644.

TANG, C. K. [1976]. “Cache design in the tightly coupled multiprocessor system,” *Proc. AFIPS National Computer Conf.,* New York (June), 749–753.

UNGER, S. H. [1958]. “A computer oriented towards spatial problems,” *Proc. Institute of Radio Enginers* 46:10 (October), 1744–1750.

WILSON, A. W., JR. [1987]. “Hierarchical cache/bus architecture for shared-memory multiprocessors,” *Proc. 14th Int’l Symposium on Computer Architecture* (June), Pittsburgh, 244–252.

WOOD, D. A. AND M. D. HILL [1995]. “Cost-effective parallel computing,” *IEEE Computer* 28:2 (February).

WULF, W. AND C. G. BELL [1972]. “C.mmp—A multi-mini-processor,” *Proc. AFIPS Fall Joint Computing Conf.* 41, part 2, 765–777.

WULF, W. AND S. P. HARBISON [1978]. “Reflections in a pool of processors—An experience report on C.mmp/Hydra,” *Proc. AFIPS 1978 National Computing Conf.* 48 (June), Anaheim, Calif., 939–951.

E X E R C I S E S

8.1 [10] <8.1> Suppose we have an application that runs in three modes: all processors used, half the processors in use, and serial mode. Assume that 0.02% of the time is serial mode, and there are 100 processors in total. Find the maximum time that can be spent in the mode when half the processors are used, if our goal is a speedup of 80.

8.2 [15] <8.1> Assume that we have a function for an application of the form F(i,p), which gives the fraction of time that exactly i processors are usable given that a total of p processors are available. This means that

$$\sum_{i=1}^{p} F(i,p) = 1$$

Assume that when i processors are in use, the application runs i times faster. Rewrite Amdahl’s Law so that it gives the speedup as a function of p for some application.

8.3 [15] <8.3> In small bus-based multiprocessors, write-through caches are sometimes used. One reason is that a write-through cache has a slightly simpler coherence protocol. Show how the basic snooping cache coherence protocol of Figure 8.12 on page 665 can be changed for a write-through cache. From the viewpoint of an implementor, what is the major hardware functionality that is not needed with a write-through cache compared with a write-back cache?

8.4 [20] <8.3> Add a clean private state to the basic snooping cache-coherence protocol (Figure 8.12 on page 665). Show the protocol in the format of Figure 8.12.

8.5 [15] <8.3> One proposed solution for the problem of false sharing is to add a valid bit per word (or even for each byte). This would allow the protocol to invalidate a word without removing the entire block, allowing a cache to keep a portion of a block in its cache while another processor wrote a different portion of the block. What extra complications are introduced into the basic snooping cache coherency protocol (Figure 8.12) if this capability is included? Remember to consider all possible protocol actions.

8.6 [12/10/15] <8.3> The performance differences for write invalidate and write update schemes can arise from both bandwidth consumption and latency. Assume a memory system with 64-byte cache blocks. Ignore the effects of contention.

a. [12] <8.3> Write two parallel code sequences to illustrate the bandwidth differences between invalidate and update schemes. One sequence should make update look much better and the other should make invalidate look much better.

b. [10] <8.3> Write a parallel code sequence to illustrate the latency advantage of an update scheme versus an invalidate scheme.

c. [15] <8.3> Show, by example, that when contention is included, the latency of update may actually be worse. Assume a bus-based machine with 50-cycle memory and snoop transactions.

8.7 [15/15] <8.3–8.4> One possible approach to achieving the scalability of distributed shared memory and the cost-effectiveness of a bus design is to combine the two approaches, using a set of nodes with memories at each node, a hybrid cache-coherence scheme, and interconnected with a bus. The argument in favor of such a design is that the use of local memories and a coherence scheme with limited broadcast results in a reduction in bus traffic, allowing the bus to be used for a larger number of processors. For these Exercises, assume the same parameters as for the Challenge bus. Assume that remote snoops and memory accesses take the same number of cycles as a memory access on the Challenge bus. Ignore the directory processing time for these Exercises. Assume that the coherency scheme works as follows on a miss: If the data are up-to-date in the local memory, it is used there. Otherwise, the bus is used to snoop for the data. Assume that local misses take 25 bus clocks.

a. [15] <8.3–8.4> Find the time for a read or write miss to data that are remote.

b. [15] <8.3–8.4> Ignoring contention and using the data from the Ocean benchmark run on 16 processors for the frequency of local and remote misses (Figure 8.26 on page 687), estimate the average memory access time versus that for a Challenge using the same total miss rate.

8.8 [20/15] <8.4> If an architecture allows a relaxed consistency model, the hardware can improve the performance of write misses by allowing the write miss to proceed immediately, buffering the write data until ownership is obtained.

a. [20] <8.4> Modify the directory protocol in Figure 8.24 on page 683 and in Figure 8.25 on page 684 to do this. Show the protocol in the same format as these two figures.

b. [15] <8.4> Assume that the write buffer is large enough to hold the write until ownership is granted, and that write ownership and any required invalidates always complete before a release is reached. If the extra time to complete a write is 100 processor clock cycles and writes generate 40% of the misses, find the performance advantage for the relaxed consistency machines versus the original protocol using the FFT data on 32 processors (Figure 8.26 on page 687).

8.9 [12/15] <8.3,8.4,8.8> Although it is widely believed that buses are the ideal interconnect for small-scale multiprocessors, this may not always be the case. For example, increases in processor performance are lowering the processor count at which a more distributed implementation becomes attractive. Because a standard bus-based implementation uses the bus both for access to memory and for interprocessor coherency traffic, it has a uniform memory access time for both. In comparison, a distributed memory implementation may sacrifice on remote memory access, but it can have a much better local memory access time.

Consider the design of a DSM multiprocessor with 16 processors. Assume the R4400 cache miss overheads shown for the Challenge design (see pages 730–731). Assume that a memory access takes 150 ns from the time the address is available from either the local processor or a remote processor until the first word is delivered.

a. [12] <8.3,8.4,8.8> How much faster is a local access than on the Challenge?

b. [15] <8.3,8.4,8.8> Assume that the interconnect is a 2D grid with links that are 16 bits wide and clocked at 100 MHz, with a start-up time of five cycles for a message. Assume one clock cycle between nodes in the network, and ignore overhead in the messages and contention (i.e., assume that the network bandwidth is not the limit). Find the average remote memory access time, assuming a uniform distribution of remote requests. How does this compare to the Challenge case? What is the largest fraction of remote misses for which the DSM machine will have a lower average memory access time than that of the Challenge machine?

8.10 [20/15/30] <8.4> One downside of a straightforward implementation of directories using fully populated bit vectors is that the total size of the directory information scales as the product: Processor count × Memory blocks. If memory is grown linearly with processor count, then the total size of the directory grows quadratically in the processor count. In practice, because the directory needs only 1 bit per memory block (which is typically 32 to 128 bytes), this problem is not serious for small to moderate processor counts. For example, assuming a 128-byte block, the amount of directory storage compared to main memory is Processor count/1024, or about 10% additional storage with 100 processors. This problem can be avoided by observing that we only need to keep an amount of information that is proportional to the cache size of each processor. We explore some solutions in these Exercises.

a. [20] <8.4> One method to obtain a scalable directory protocol is to organize the machine as a logical hierarchy with the processors at the leaves of the hierarchy and directories positioned at the root of each subtree. The directory at each subtree root records which descendents cache which memory blocks, as well as which memory blocks with a home in that subtree are cached outside of the subtree. Compute the amount of storage needed to record the processor information for the directories, assuming that each directory is fully associative. Your answer should incorporate both the number of nodes at each level of the hierarchy as well as the total number of nodes.

b. [15] <8.4> Assume that each level of the hierarchy in part (a) has a lookup cost of 50 cycles plus a cost to access the data or cache of 50 cycles, when the point is reached. We want to compute the AMAT (average memory access time—see Chapter 5) for a 64-processor machine with four-node subtrees. Use the data from the Ocean benchmark run on 64 processors (Figure 8.26) and assume that all noncoherence misses occur within a subtree node and that coherence misses are uniformly distributed across the machine. Find the AMAT for this machine. What does this say about hierarchies?

c. [30] <8.4> An alternative approach to implementing directory schemes is to implement bit vectors that are not dense. There are two such strategies: one reduces the number of bit vectors needed and the other reduces the number of bits per vector. Using traces, you can compare these schemes. First, implement the directory as a four-way set-associative cache storing full bit vectors, but only for the blocks that are cached outside of the home node. If a directory cache miss occurs, choose a directory entry and invalidate the entry. Second, implement the directory so that every entry has 8 bits. If a block is cached in only one node outside of its home, this field contains the node number. If the block is cached in more than one node outside its home, this field is a bit vector with each bit indicating a group of eight processors, at least one of which caches the block. Using traces of 64-processor execution, simulate the behavior of these two schemes. Assume a perfect cache for nonshared references, so as to focus on coherency behavior. Determine the number of extraneous invalidations as the directory cache size is increased.

8.11 [25/40] <8.7> Prefetching and relaxed consistency models are two methods of tolerating the latency of longer access in multiprocessors. Another scheme, originally used in the HEP multiprocessor and incorporated in the MIT Alewife multiprocessor, is to switch to another activity when a long-latency event occurs. This idea, called *multiple context* or *multithreading,* works as follows:

- The processor has several register files and maintains several PCs (and related program states). Each register file and PC holds the program state for a separate parallel thread.
- When a long-latency event occurs, such as a cache miss, the processor switches to another thread, executing instructions from that thread while the miss is being handled.

a. [25] <8.7> Using the data for the Ocean benchmark running on 64 processors (Figure 8.26), determine how many contexts are needed to hide all the latency of remote accesses. Assume that local cache misses take 40 cycles and that remote misses take 120 cycles. Assume that the increased demands due to a higher request rate do not affect either the latency or the bandwidth of communications.

b. [40] <8.7> Implement a simulator for a multiple-context directory-based machine. Use the simulator to evaluate the performance gains from multiple context. How significant are contention and the added bandwidth demands in limiting the gains?

8.12 [25] <8.7> Prove that in a two-level cache hierarchy, where L1 is closer to the processor, inclusion is maintained with no extra action if L2 has at least as much associativity as L1, both caches use LRU replacement, and both caches have the same block size.

8.13 [20] <8.4,8.9> As we saw in *Fallacies and Pitfalls,* data distribution can be important when an application has a nontrivial private data miss rate caused by capacity misses. This problem can be attacked with compiler technology (distributing the data in blocks) or through architectural support. One architectural technique is called cache-only memory architecture (COMA), a version of which was implemented in the KSR-1. The basic idea in COMA is to make the distributed memories into caches, so that blocks *can* be replicated and migrated at the memory level of the hierarchy, as well as in higher levels. Thus, a COMA architecture can change what would be remote capacity misses on a DSM architecture into local capacity misses, by creating copies of data in the local memory. This hardware capability allows the software to ignore the initial distribution of data to different memories. The hardware required to implement a cache in the local memory will usually lead to a slight increase in the memory access time of the memory on a COMA architecture.

Assume that we have a DSM and a COMA machine where remote coherence misses are uniformly distributed and take 100 clocks. Assume that all capacity misses on the COMA machine hit in the local memory and require 50 clock cycles. Assume that capacity misses take 40 cycles when they are local on the DSM machine and 75 cycles otherwise. Using the Ocean data for 32 processors (Figure 8.13), find what fraction of the capacity misses on the DSM machine must be local if the performance of the two machines is identical.

8.14 [15] <8.5> Some machines have implemented a special broadcast coherence protocol just for locks, sometimes even using a different bus. Evaluate the performance of the spin lock in the Example on page 699 assuming a write broadcast protocol.

8.15 [15] <8.5> Implement the barrier in Figure 8.34 on page 701, using queuing locks. Compare the performance to the spin-lock barrier.

8.16 [15] <8.5> Implement the barrier in Figure 8.34 on page 701, using fetch-and-increment. Compare the performance to the spin-lock barrier.

8.17 [15] <8.5> Implement the barrier on page 705, so that barrier release is also done with a combining tree.

8.18 [28] <8.6> Write a set of programs so that you can distinguish the following consistency models: sequential consistency, processor consistency or total store order, partial store order, weak ordering, and release consistency. Using multiprocessors that you have access to, determine what consistency model different machines support. Note that, because of timing, you may need to try accesses multiple times to uncover all orderings allowed by a machine.

8.19 [30] <8.3–8.5> Using an available shared-memory multiprocessor, see if you can determine the organization and latencies of its memory hierarchy. For each level of the hierarchy, you can look at the total size, block size, and associativity, as well as the latency of each level of the hierarchy. If the machine uses a nonbus interconnection network, see if you can discover the topology, latency, and bandwidth characteristics of the network.

8.20 [20] <8.4> As we discussed earlier, the directory controller can send invalidates for lines that have been replaced by the local cache controller. To avoid such messages, and to keep the directory consistent, replacement hints are used. Such messages tell the controller that a block has been replaced. Modify the directory coherence protocol of section 8.4 to use such replacement hints.

8.21 [25] <8.6> Prove that for synchronized programs, a release consistency model allows only the same results as sequential consistency.

8.22 [15] <8.5> Find the time for *n* processes to synchronize using a standard barrier. Assume that the time for a single process to update the count and release the lock is *c*.

8.23 [15] <8.5> Find the time for *n* processes to synchronize using a combining tree barrier. Assume that the time for a single process to update the count and release the lock is *c*.

8.24 [25] <8.5> Implement a software version of the queuing lock for a bus-based system. Using the model in the Example on page 699, how long does it take for 20 processors to acquire and release the lock? You need only count bus cycles.

8.25 [20/30] <8.2–8.5> Both researchers and industry designers have explored the idea of having the capability to explicitly transfer data between memories. The argument in favor of such facilities is that the programmer can achieve better overlap of computation and communication by explicitly moving data when it is available. The first part of this exercise explores the potential on paper; the second explores the use of such facilities on real machines.

a. [20] <8.2–8.5> Assume that cache misses stall the processor, and that block transfer occurs into the local memory of a DSM node. Assume that remote misses cost 100 cycles and that local misses cost 40 cycles. Assume that each DMA transfer has an overhead of 10 cycles. Assuming that all the coherence traffic can be replaced with DMA into main memory followed by a cache miss, find the potential improvement for Ocean running on 64 processors (Figure 8.26).

b. [30] <8.2–8.5> Find a machine that implements both shared memory (coherent or incoherent) and a simple DMA facility. Implement a blocked matrix multiply using only shared memory and using the DMA facilities with shared memory. Is the latter faster? How much? What factors make the use of a block data transfer facility attractive?

8.26 [Discussion] <8.8> Construct a scenario whereby a truly revolutionary architecture—pick your favorite candidate—will play a significant role. *Significant* is defined as 10% of the computers sold, 10% of the users, 10% of the money spent on computers, or 10% of some other figure of merit.

8.27 [40] <8.2,8.7,8.9> A multiprocessor or multicomputer is typically marketed using programs that can scale performance linearly with the number of processors. The project here is to port programs written for one machine to the others and to measure their absolute performance and how it changes as you change the number of processors. What changes need to be made to improve performance of the ported programs on each machine? What is the ratio of processor performance according to each program?

8.28 [35] <8.2,8.7,8.9> Instead of trying to create fair benchmarks, invent programs that make one multiprocessor or multicomputer look terrible compared with the others, and also programs that always make one look better than the others. It would be an interesting result if you couldn't find a program that made one multiprocessor or multicomputer look worse than the others. What are the key performance characteristics of each organization?

8.29 [40] <8.2,8.7,8.9> Multiprocessors and multicomputers usually show performance increases as you increase the number of processors, with the ideal being *n* times speedup for *n* processors. The goal of this biased benchmark is to make a program that gets worse performance as you add processors. For example, this means that one processor on the multiprocessor or multicomputer runs the program fastest, two are slower, four are slower than two, and so on. What are the key performance characteristics for each organization that give inverse linear speedup?

8.30 [50] <8.2,8.7,8.9> Networked workstations can be considered multicomputers, albeit with somewhat slower, though perhaps cheaper, communication relative to computation. Port multicomputer benchmarks to a network using remote procedure calls for communication. How well do the benchmarks scale on the network versus the multicomputer? What are the practical differences between networked workstations and a commercial multicomputer?

Blank