4

Multiprocessors and Thread-Level Parallelism

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.

W. Jack Bouknight et al.
The Illiac IV System (1972)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
describing Intel's future direction at the Intel Developers Forum in 2005

4.1 Introduction

As the quotation that opens this chapter shows, the view that advances in uniprocessor architecture were nearing an end has been held by some researchers for many years. Clearly these views were premature; in fact, during the period of 1986–2002, uniprocessor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s.

Nonetheless, the importance of multiprocessors was growing throughout the 1990s as designers sought a way to build servers and supercomputers that achieved higher performance than a single microprocessor, while exploiting the tremendous cost-performance advantages of commodity microprocessors. As we discussed in Chapters 1 and 3, the slowdown in uniprocessor performance arising from diminishing returns in exploiting ILP, combined with growing concern over power, is leading to a new era in computer architecture—an era where multiprocessors play a major role. The second quotation captures this clear inflection point.

This trend toward more reliance on multiprocessing is reinforced by other factors:

- A growing interest in servers and server performance
- A growth in data-intensive applications
- The insight that increasing performance on the desktop is less important (outside of graphics, at least)
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant natural thread-level parallelism
- The advantages of leveraging a design investment by replication rather than unique design—all multiprocessor designs provide such leverage

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 youth, with ideas coming and going and, until very recently, more architectures failing than succeeding. Full coverage of the multiprocessor design space and its trade-offs would require another volume. (Indeed, Culler, Singh, and Gupta [1999] cover *only* multiprocessors in their 1000-page book!) Second, broad 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: multiprocessors with small to medium numbers of processors (4 to 32). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (32 or more processors), primarily in Appendix H, which covers more aspects of the design of such processors, as well as the behavior performance for parallel scientific work-

loads, a primary class of applications for large-scale multiprocessors. In the large-scale multiprocessors, the interconnection networks are a critical part of the design; Appendix E focuses on that topic.

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 Appendix K.

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 40 years ago, Flynn [1966] 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 multiprocessor, and placed all computers into one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. SIMD computers exploit *data-level parallelism* by applying the same operations to multiple items of data in parallel. 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. For applications that display significant data-level parallelism, the SIMD approach can be very efficient. The multimedia extensions discussed in Appendices B and C are a form of SIMD parallelism. Vector architectures, discussed in Appendix F, are the largest class of SIMD architectures. SIMD approaches have experienced a rebirth in the last few years with the growing importance of graphics performance, especially for the game market. SIMD approaches are the favored method for achieving the high performance needed to create realistic three-dimensional, real-time virtual environments.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date.
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data. MIMD computers exploit *thread-level parallelism*, since multiple threads operate in parallel. In general, thread-level parallelism is more flexible than data-level parallelism and thus more generally applicable.

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

Because the MIMD model can exploit thread-level parallelism, it is the architecture of choice for general-purpose multiprocessors and our focus in this chapter. Two other factors have also contributed to the rise of the MIMD multiprocessors:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.
2. MIMDs can build on the cost-performance advantages of off-the-shelf processors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers. Furthermore, multicore chips leverage the design investment in a single processor core by replicating it.

One popular class of MIMD computers are *clusters*, which often use standard components and often standard network technology, so as to leverage as much commodity technology as possible. In Appendix H we distinguish two different types of clusters: *commodity clusters*, which rely entirely on third-party processors and interconnection technology, and *custom clusters*, in which a designer customizes either the detailed node design or the interconnection network, or both.

In a commodity cluster, the nodes of a cluster are often blades or rack-mounted servers (including small-scale multiprocessor servers). Applications that focus on throughput and require almost no communication among threads, such as Web serving, multiprogramming, and some transaction-processing applications, can be accommodated inexpensively on a cluster. Commodity clusters are often assembled by users or computer center directors, rather than by vendors.

Custom clusters are typically focused on parallel applications that can exploit large amounts of parallelism on a single problem. Such applications require a significant amount of communication during the computation, and customizing the node and interconnect design makes such communication more efficient than in a commodity cluster. Currently, the largest and fastest multiprocessors in existence are custom clusters, such as the IBM Blue Gene, which we discuss in Appendix H.

Starting in the 1990s, the increasing capacity of a single chip allowed designers to place multiple processors on a single die. This approach, initially called *on-chip multiprocessing* or *single-chip multiprocessing*, has come to be called *multicore*, a name arising from the use of multiple processor cores on a single die. In such a design, the multiple cores typically share some resources, such as a second- or third-level cache or memory and I/O buses. Recent processors, including the IBM Power5, the Sun T1, and the Intel Pentium D and Xeon-MP, are multicore and multithreaded. Just as using multiple copies of a microprocessor in a multiprocessor leverages a design investment through replication, a multicore achieves the same advantage relying more on replication than the alternative of building a wider superscalar.

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. A *process* is a segment of code that may be run independently; the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of other processes.

It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*. Today, the term *thread* is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space. For example, a multithreaded architecture actually allows the simultaneous execution of multiple processes, with potentially separate address spaces, as well as multiple threads that share the same address space.

To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute. The independent threads within a single process are typically identified by the programmer or created by the compiler. The threads may come from large-scale, independent processes scheduled and manipulated by the operating system. At the other extreme, a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop. Although the amount of computation assigned to a thread, called the *grain size*, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor (see Appendix F) may be able to efficiently parallelize operations on short vectors, the resulting grain size when the parallelism is split among many threads may be so small that the overhead makes the exploitation of the parallelism prohibitively expensive.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. We refer to the multiprocessors 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*, has at most a few dozen processor chips (and less than 100 cores) in 2006. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory. With large caches, a single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By using multiple point-to-point connections, or a switch, and adding additional memory banks, a centralized shared-memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable,

sharing a centralized memory becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are most often called *symmetric (shared-memory) multiprocessors* (SMPs), and this style of architecture is sometimes called *uniform memory access* (UMA), arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks. Figure 4.1 shows what these multiprocessors look like. This type of symmetric shared-memory architecture is currently by far the most popular organization. The architecture of such multiprocessors is the topic of Section 4.2.

The second group consists of multiprocessors with physically distributed memory. Figure 4.2 shows what these multiprocessors look like. 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 without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The larger number of processors also raises the need for a high-bandwidth interconnect, of which we will see examples in Appendix E.

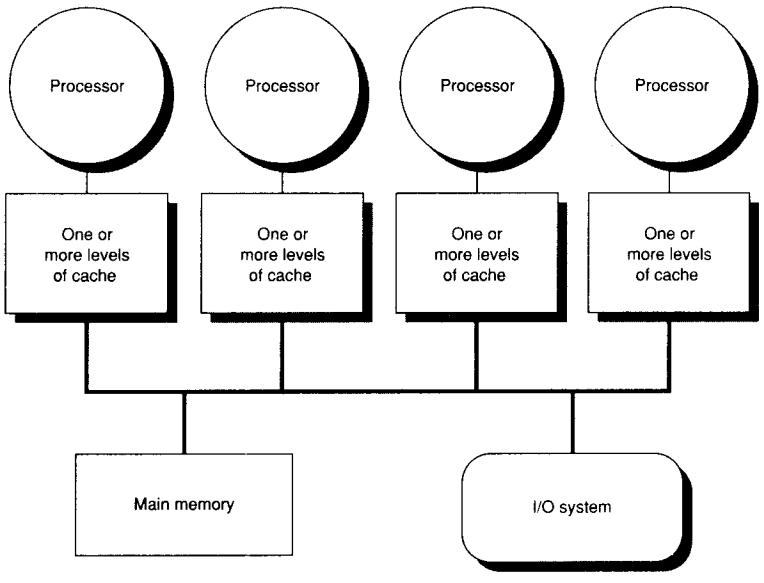

Figure 4.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by one or more buses or a switch. The key architectural property is the uniform access time to all of memory from all the processors.

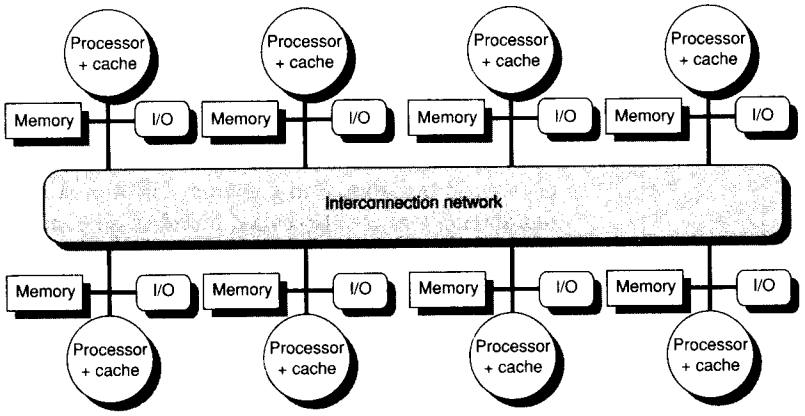

Figure 4.2 The basic architecture of a distributed-memory multiprocessor consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

Both direction networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used.

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 disadvantages for a distributed-memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories. As we will see shortly, the use of distributed memory also leads to two different paradigms for interprocessor communication.

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.

In the first method, communication occurs through a shared address space, as it does in a symmetric shared-memory architecture. 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 multiprocessors are called *distributed shared memory* (DSM) 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 symmetric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors are also called NUMAs (nonuniform 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 multiprocessors, 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. Initially, such computers were built with different processing nodes and specialized interconnection networks. Today, most designs of this type are actually clusters, which we discuss in Appendix H.

With each of these organizations for the address space, there is an associated communication mechanism. For a multiprocessor 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 multiprocessors. For a multiprocessor with multiple address spaces, communication of data is done by explicitly passing messages among the processors. Therefore, these multiprocessors are often called *message-passing multiprocessors*. Clusters inherently use message passing.

Challenges of Parallel Processing

The application of multiprocessors ranges from running independent tasks with essentially no communication to running parallel programs where threads must communicate to complete the task. Two important hurdles, both explainable with Amdahl's Law, make parallel processing challenging. The degree to which these hurdles are difficult or easy is determined both by the application and by the architecture.

The first hurdle 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 any parallel processor, 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 previous equation:

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

Simplifying this equation yields

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

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. In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode.

The second major challenge in parallel processing involves the large latency of remote access in a parallel processor. In existing shared-memory multiprocessors, communication of data between processors may cost anywhere from 50 clock cycles (for multicores) to over 1000 clock cycles (for large-scale multiprocessors), depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. The effect of long communication delays is clearly substantial. Let's consider a simple example.

Example Suppose we have an application running on a 32-processor multiprocessor, which has a 200 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 is slightly optimistic. Processors are stalled on a remote request, and the processor clock rate is 2 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

Answer It is simpler to first calculate the CPI. The effective CPI for the multiprocessor with 0.2% remote references is

$$\begin{aligned}\text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost}\\ &= 0.5 + 0.2\% \times \text{Remote request cost}\end{aligned}$$

The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{200 \text{ ns}}{0.5 \text{ ns}} = 400 \text{ cycles}$$

Hence we can compute the CPI:

$$\text{CPI} = 0.5 + 0.8 = 1.3$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 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 performance 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 software mechanisms, such as restructuring the data to make more accesses local. We can try to tolerate the latency by using multithreading (discussed in Chapter 3 and later in this chapter) or by using prefetching (a topic we cover extensively in Chapter 5).

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, Sections 4.2 and 4.3 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 4.5 discusses synchronization, which, because it inherently involves interprocessor communication and also can limit parallelism, is a major potential bottleneck. Section 4.6 covers latency-hiding techniques and memory consistency models for shared memory. In Appendix I, we focus primarily on large-scale multiprocessors, which are used predominantly for scientific work. In that appendix, we examine the nature of such applications and the challenges of achieving speedup with dozens to hundreds of processors.

Understanding a modern shared-memory multiprocessor requires a good understanding of the basics of caches. Readers who have covered this topic in our introductory book, *Computer Organization and Design: The Hardware/Software Interface*, will be well-prepared. If topics such as write-back caches and multilevel caches are unfamiliar to you, you should take the time to review Appendix C.

4.2 Symmetric Shared-Memory Architectures

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 symmetric multiprocessors were extremely cost-effective, provided that a sufficient amount of memory bandwidth existed. Early designs of such multiprocessors were able to place the processor and cache subsystem on a board, which plugged into the bus backplane. Subsequent versions of such designs in the 1990s could achieve higher densities with two to four processors per board, and often used multiple buses and interleaved memories to support the faster processors.

IBM introduced the first on-chip multiprocessor for the general-purpose computing market in 2000. AMD and Intel followed with two-processor versions for the server market in 2005, and Sun introduced T1, an eight-processor multicore in 2006. Section 4.8 looks at the design and performance of T1. The earlier Figure 4.1 on page 200 shows a simple diagram of such a multiprocessor. With the more recent, higher-performance processors, the memory demands have outstripped the capability of reasonable buses. As a result, most recent designs use a small-scale switch or a limited point-to-point network.

Symmetric shared-memory machines usually support the caching of both shared and private data. *Private data* are used by a single processor, while *shared data* are 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?

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. Figure 4.3 illustrates the problem and shows how two different processors

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 4.3 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!

can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem.*

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, although 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 in time 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 ensure 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 see why, 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 4.6.

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 now, make the following two assumptions. First, a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write. Second, the processor does not change the order of any write with respect to any other memory access. These two conditions mean that if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A. These restrictions allow 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 4.6, where we will see exactly the implications of this definition, as well as the alternatives.

Basic Schemes for Enforcing Coherence

The coherence problem for multiprocessors and I/O, although 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 normally 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 migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

Coherent caches also provide replication for shared data that are 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 4.4, when we discuss scalable shared-memory architecture. Directory-based coherence has slightly higher implementation overhead than snooping, but it can scale to larger processor counts. The Sun T1 design, the topic of Section 4.8, uses directories, albeit with a central physical memory.
- *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, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or switch), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access. 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. In the following section we explain snoop-based cache coherence as implemented with a shared bus, but any communication medium that broadcasts cache misses to all processors can be used to implement a snooping-based coherence scheme. This broadcasting to all caches is what makes snooping protocols simple to implement but also limits their scalability.

Snooping Protocols

There are two ways to maintain the coherence requirement described in the prior 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.

Figure 4.4 shows an example of an invalidation protocol for a snooping bus with write-back caches in action. To see how this protocol 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

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 4.4 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 update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it.

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.

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. Because a write update protocol must broadcast all writes to shared cache lines, it consumes considerably more bandwidth. For this reason, all recent multiprocessors have opted to implement a write invalidate protocol, and 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 multiprocessor is the use of the bus, or another broadcast medium, 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 are invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors attempt to write

shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes. One implication of this scheme is that a write to a shared data item cannot actually complete until it obtains bus access. All coherence schemes require some method of serializing accesses to the same cache block, either by serializing access to the communication medium or another shared structure.

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 the next chapter.) In a design with adequate memory bandwidth to support the write traffic from the processors, using write through simplifies the implementation of cache coherence.

For a write-back cache, 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 cache 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. The additional complexity comes from having to retrieve the cache block from a processor's cache, which can often take longer than retrieving it from the shared memory if the processors are in separate chips. Since write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors and have been the approach chosen in most multiprocessors, despite the additional complexity of maintaining coherence. Therefore, we will examine the implementation of coherence with write-back caches.

The normal cache tags can be used to implement the process of snooping, and 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 *exclusive*. 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.

Every bus transaction must check the cache-address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags. The interference can also be reduced in a multilevel cache by directing the snoop requests to the L2 cache, which the processor uses only when it has a miss in the L1 cache. For this scheme to work, every entry in the L1 cache must be present in the L2 cache, a property called the *inclusion property*. If the snoop gets a hit in the L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the processor and the snooping activity. We discuss the inclusion property in more detail in the next chapter.

An Example Protocol

A snooping 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 (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Logically, you can think of a separate controller being associated with each block; that is, snooping operations or cache requests for different blocks can proceed independently. In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion (that is, one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). Also, remember that, although we refer to a bus in the following description, any interconnection network that supports a broadcast to all the coherence controllers and their associated caches can be used to implement snooping.

The simple protocol we consider has three states: invalid, shared, and modified. The shared state indicates that the block is potentially shared, while the modified state indicates that the block has been updated in the cache; note that the modified state *implies* that the block is exclusive. Figure 4.5 shows the requests generated by the processor-cache module in a node (in the top half of the table) as well as those coming from the bus (in the bottom half of the table). This protocol is for a write-back cache but is easily changed to work for a write-through cache by reinterpreting the modified state as an exclusive state and updating the cache on writes in the normal fashion for a write-through cache. The most common extension of this basic protocol is the addition of an exclusive

state, which describes a block that is unmodified but held in only one cache; the caption of Figure 4.5 describes this state and its addition in more detail.

When an invalidate or a write miss is placed on the bus, any processors with copies of the cache block invalidate it. For a write-through cache, the data for a write miss can always be retrieved from the memory. For a write miss in a write-back cache, if the block is exclusive in just one cache, that cache also writes back the block; otherwise, the data can be read from memory.

Figure 4.6 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 processor requests (on the left, which corresponds to the top half of the table in Figure 4.5), as opposed to transitions based on bus requests (on the right, which corresponds to the bottom half of the table in Figure 4.5). 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. Most of the state changes indicated by arcs in the left half of Figure 4.6 would be needed in a write-back uniprocessor cache, with the exception being the invalidate on a write hit to a shared block. The state changes represented by the arcs in the right half of Figure 4.6 are needed only for coherence and would not appear at all in a uniprocessor cache controller.

As mentioned earlier, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. Figure 4.7 shows how the state transitions in the right half of Figure 4.6 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 one or more 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 an invalidate or 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 cache with the exclusive copy changes its state to shared.

The actions in gray in Figure 4.7, 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, which simplifies the implementation.

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

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	processor	shared or modified	normal hit	Read data in cache.
Read miss	processor	invalid	normal miss	Place read miss on bus.
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.
Read miss	processor	modified	replacement	Address conflict miss: write back block, then place read miss on bus.
Write hit	processor	modified	normal hit	Write data in cache.
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	processor	invalid	normal miss	Place write miss on bus.
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.
Write miss	processor	modified	replacement	Address conflict miss: write back block, then place write miss on bus.
Read miss	bus	shared	no action	Allow memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid.

Figure 4.5 The cache coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request. The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read, misses, write misses, or invalidates snooped from the bus, an action is required *only* if the read or write addresses match a block in the cache and the block is valid. Some protocols also introduce a state to designate when a block is exclusively in one cache but has not yet been written. This state can arise if a write access is broken into two pieces: getting the block exclusively in one cache and then subsequently updating it; in such a protocol this "exclusive unmodified state" is transient, ending as soon as the write is completed. Other protocols use and maintain an exclusive state for an unmodified block. In a snooping protocol, this state can be entered when a processor reads a block that is not resident in any other cache. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same processor need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a bock is modified. The popular MESI protocol, which is named for the four states it includes (modified, exclusive, shared, and invalid), uses this structure. The MOESI protocol introduces another extension: the "owned" state, as described in the caption of Figure 4.4.

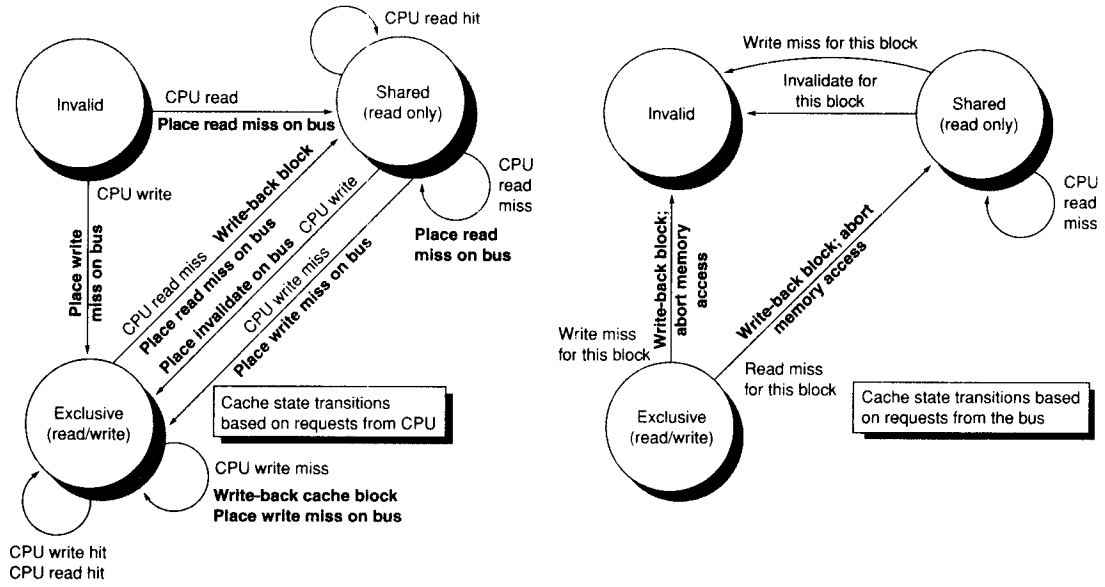

Figure 4.6 A write invalidate, cache coherence protocol for a write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the processor without a state transition shown in parentheses under the name of the 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 block 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 processor 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 processor 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 generates an invalidate. 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. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.

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 switch, as all recent multiprocessors do, then even read misses would also not be atomic.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. We will explore how these protocols are implemented without a bus shortly.

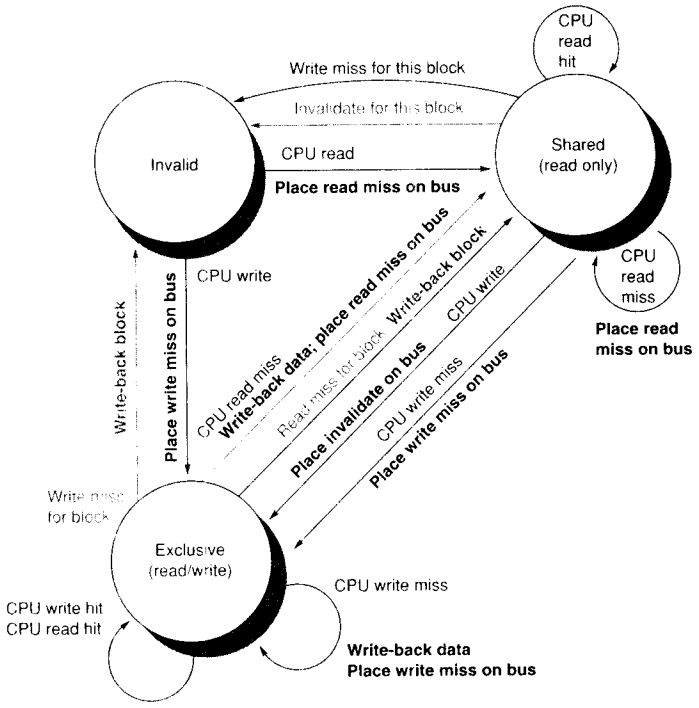

Figure 4.7 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 4.6, the activities on a transition are shown in bold.

Constructing small-scale (two to four processors) multiprocessors has become very easy. For example, the Intel Pentium 4 Xeon and AMD Opteron processors are designed for use in cache-coherent multiprocessors and have an external interface that supports snooping and allows two to four processors to be directly connected. They also have larger on-chip caches to reduce bus utilization. In the case of the Opteron processors, the support for interconnecting multiple processors is integrated onto the processor chip, as are the memory interfaces. In the case of the Intel design, a two-processor system can be built with only a few additional external chips to interface with the memory system and I/O. Although these designs cannot be easily scaled to larger processor counts, they offer an extremely cost-effective solution for two to four processors.

The next section examines the performance of these protocols for our parallel and multiprogrammed workloads; the value of these extensions to a basic protocol will be clear when we examine the performance. But before we do that, let's take a brief look at the limitations on the use of a symmetric memory structure and a snooping coherence scheme.

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

As the number of processors in a multiprocessor grows, or as the memory demands of each processor grow, any centralized resource in the system can become a bottleneck. In the simple case of a bus-based multiprocessor, the bus must support both the coherence traffic as well as normal memory traffic arising from the caches. Likewise, if there is a single memory unit, it must accommodate all processor requests. As processors have increased in speed in the last few years, the number of processors that can be supported on a single bus or by using a single physical memory unit has fallen.

How can a designer increase the memory bandwidth to support either more or faster processors? To increase the communication bandwidth between processors and memory, designers have used multiple buses as well as interconnection networks, such as crossbars or small point-to-point networks. In such designs, the memory system can be configured into multiple physical banks, so as to boost the effective memory bandwidth while retaining uniform access time to memory. Figure 4.8 shows this approach, which represents a midpoint between the two approaches we discussed in the beginning of the chapter: centralized shared memory and distributed shared memory.

The AMD Opteron represents another intermediate point in the spectrum between a snoopy and a directory protocol. Memory is directly connected to each dual-core processor chip, and up to four processor chips, eight cores in total, can be connected. The Opteron implements its coherence protocol using the point-to-point links to broadcast up to three other chips. Because the interprocessor links are not shared, the only way a processor can know when an invalid operation has

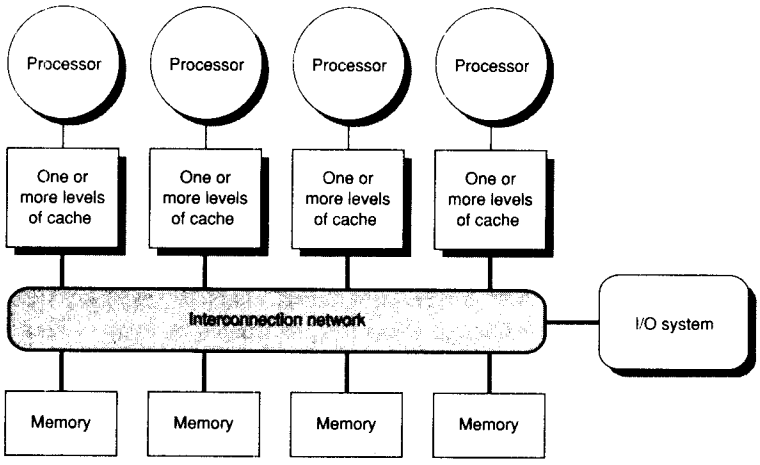

Figure 4.8 A multiprocessor with uniform memory access using an interconnection network rather than a bus.

completed is by an explicit acknowledgment. Thus, the coherence protocol uses a broadcast to find potentially shared copies, like a snoopy protocol, but uses the acknowledgments to order operations, like a directory protocol. Interestingly, the remote memory latency and local memory latency are not dramatically different, allowing the operating system to treat an Opteron multiprocessor as having uniform memory access.

A snoopy cache coherence protocol can be used without a centralized bus, but still requires that a broadcast be done to snoop the individual caches on every miss to a potentially shared cache block. This cache coherence traffic creates another limit on the scale and the speed of the processors. Because coherence traffic is unaffected by larger caches, faster processors will inevitably overwhelm the network and the ability of each cache to respond to snoop requests from *all* the other caches. In Section 4.4, we examine directory-based protocols, which eliminate the need for broadcast to all caches on a miss. As processor speeds and the number of cores per processor increase, more designers are likely to opt for such protocols to avoid the broadcast limit of a snoopy protocol.

Implementing Snoopy Cache Coherence

The devil is in the details.

Classic proverb

When we wrote the first edition of this book in 1990, our final "Putting It All Together" was a 30-processor, single bus multiprocessor using snoop-based coherence; the bus had a capacity of just over 50 MB/sec, which would not be enough bus bandwidth to support even one Pentium 4 in 2006! When we wrote the second edition of this book in 1995, the first cache coherence multiprocessors with more than a single bus had recently appeared, and we added an appendix describing the implementation of snooping in a system with multiple buses. In 2006, *every* multiprocessor system with more than two processors uses an interconnect other than a single bus, and designers must face the challenge of implementing snooping without the simplification of a bus to serialize events.

As we said earlier, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss, communicating with the other processors and memory, getting the most recent value for a write miss and ensuring that any invalidates are processed, and updating the cache cannot be done as if they took a single cycle.

In a simple single-bus system, these steps can be made effectively atomic by arbitrating for the bus first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In most bus-based multiprocessors a single line is used to signal when all necessary invalidates have been received and are being processed. Following that signal, the processor that generated the miss can release the bus,

knowing that any required actions will be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively makes the individual steps atomic.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: one write is processed and precedes before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a snoopy system ensuring that a race has only one winner is ensured by using broadcast for all misses as well as some basic properties of the interconnection network. These properties, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snoopy cache coherence without a bus. We explain the details in Appendix H.

4.3 Performance of Symmetric Shared-Memory Multiprocessors

In a multiprocessor using a snoopy coherence 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.

Appendix C breaks the uniprocessor miss rate into the three C's classification (capacity, compulsory, and conflict) and provides insight into both application behavior and potential improvements to the cache design. Similarly, the misses that arise from interprocessor communication, which are often called *coherence misses,* can be broken into two separate sources.

The first source is the so-called *true sharing misses* that arise from the communication of data through the cache coherence mechanism. In an invalidation-based protocol, the first write by a processor to a shared cache block causes an invalidation to establish ownership of that block. Additionally, when another processor attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred. Both these misses are classified as true sharing misses since they directly arise from the sharing of data among processors.

The second effect, called *false sharing,* arises from the use of an invalidation-based coherence algorithm with a single valid bit per cache 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. 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, which is in the shared state in the caches of both P1 and P2. 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. The cache block containing x1 will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols this will be handled as an *upgrade request*, which generates a bus invalidate, but does not transfer the cache block.
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.

Although we will see the effects of true and false sharing misses in commercial workloads, the role of coherence misses is more significant for tightly coupled applications that share significant amounts of user data. We examine their effects in detail in Appendix H, when we consider the performance of a parallel scientific workload.

A Commercial Workload

In this section, we examine the memory system behavior of a four-processor shared-memory multiprocessor. The results were collected either on an AlphaServer 4100 or using a configurable simulator modeled after the AlphaServer 4100. Each processor in the AlphaServer 4100 is an Alpha 21164, which issues up to four instructions per clock and runs at 300 MHz. Although the clock rate of the Alpha processor in this system is considerably slower than processors in recent systems, the basic structure of the system, consisting of a four-issue processor and a three-level cache hierarchy, is comparable to many recent systems. In particular, each processor has a three-level cache hierarchy:

- L1 consists of a pair of 8 KB direct-mapped on-chip caches, one for instruction and one for data. The block size is 32 bytes, and the data cache is write through to L2, using a write buffer.
- L2 is a 96 KB on-chip unified three-way set associative cache with a 32-byte block size, using write back.
- L3 is an off-chip, combined, direct-mapped 2 MB cache with 64-byte blocks also using write back.

The latency for an access to L2 is 7 cycles, to L3 it is 21 cycles, and to main memory it is 80 clock cycles (typical without contention). Cache-to-cache transfers, which occur on a miss to an exclusive block held in another cache, require 125 clock cycles. Although these miss penalties are smaller than today's higher clock systems would experience, the caches are also smaller, meaning that a more recent system would likely have lower miss rates but higher miss penalties.

The workload used for this study consists of three applications:

1. An online transaction-processing workload (OLTP) modeled after TPC-B (which has similar memory behavior to its newer cousin TPC-C) and using Oracle 7.3.2 as the underlying database. The workload consists of a set of client processes that generate requests and a set of servers that handle them. The server processes consume 85% of the user time, with the remaining going to the clients. Although the I/O latency is hidden by careful tuning and enough requests to keep the CPU busy, the server processes typically block for I/O after about 25,000 instructions.
2. A decision support system (DSS) workload based on TPC-D and also using Oracle 7.3.2 as the underlying database. The workload includes only 6 of the 17 read queries in TPC-D, although the 6 queries examined in the benchmark span the range of activities in the entire benchmark. To hide the I/O latency, parallelism is exploited both within queries, where parallelism is detected during a query formulation process, and across queries. Blocking calls are much less frequent than in the OLTP benchmark; the 6 queries average about 1.5 million instructions before blocking.

Benchmark	% time user mode	% time kernel	% time CPU idle
OLTP	71	18	11
DSS (average across all queries)	87	4	9
AltaVista	> 98	< 1	< 1

Figure 4.9 The distribution of execution time in the commercial workloads. The OLTP benchmark has the largest fraction of both OS time and CPU idle time (which is I/O wait time). The DSS benchmark shows much less OS time, since it does less I/O, but still more than 9% idle time. The extensive tuning of the AltaVista search engine is clear in these measurements. The data for this workload were collected by Barroso et al. [1998] on a four-processor AlphaServer 4100.

3. A Web index search (AltaVista) benchmark based on a search of a memory-mapped version of the AltaVista database (200 GB). The inner loop is heavily optimized. Because the search structure is static, little synchronization is needed among the threads.

The percentages of time spent in user mode, in the kernel, and in the idle loop are shown in Figure 4.9. The frequency of I/O increases both the kernel time and the idle time (see the OLTP entry, which has the largest I/O-to-computation ratio). AltaVista, which maps the entire search database into memory and has been extensively tuned, shows the least kernel or idle time.

Performance Measurements of the Commercial Workload

We start by looking at the overall CPU execution for these benchmarks on the four-processor system; as discussed on page 220, these benchmarks include substantial I/O time, which is ignored in the CPU time measurements. We group the six DSS queries as a single benchmark, reporting the average behavior. The effective CPI varies widely for these benchmarks, from a CPI of 1.3 for the AltaVista Web search, to an average CPI of 1.6 for the DSS workload, to 7.0 for the OLTP workload. Figure 4.10 shows how the execution time breaks down into instruction execution, cache and memory system access time, and other stalls (which are primarily pipeline resource stalls, but also include TLB and branch mispredict stalls). Although the performance of the DSS and AltaVista workloads is reasonable, the performance of the OLTP workload is very poor, due to a poor performance of the memory hierarchy.

Since the OLTP workload demands the most from the memory system with large numbers of expensive L3 misses, we focus on examining the impact of L3 cache size, processor count, and block size on the OLTP benchmark. Figure 4.11 shows the effect of increasing the cache size, using two-way set associative caches, which reduces the large number of conflict misses. The execution time is improved as the L3 cache grows due to the reduction in L3 misses. Surprisingly,

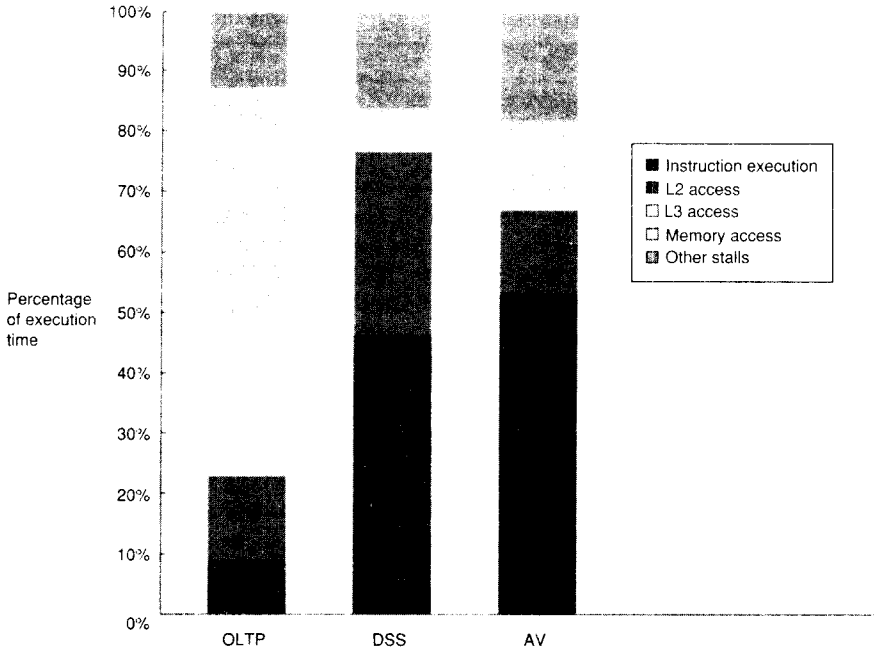

Figure 4.10 The execution time breakdown for the three programs (OLTP, DSS, and AltaVista) in the commercial workload. The DSS numbers are the average across six different queries. The CPI varies widely from a low of 1.3 for AltaVista, to 1.61 for the DSS queries, to 7.0 for OLTP. (Individually, the DSS queries show a CPI range of 1.3 to 1.9.) Other stalls includes resource stalls (implemented with replay traps on the 21164), branch mispredict, memory barrier, and TLB misses. For these benchmarks, resource-based pipeline stalls are the dominant factor. These data combine the behavior of user and kernel accesses. Only OLTP has a significant fraction of kernel accesses, and the kernel accesses tend to be better behaved than the user accesses! All the measurements shown in this section were collected by Barroso, Gharachorloo, and Bugnion [1998].

almost all of the gain occurs in going from 1 to 2 MB, with little additional gain beyond that, despite the fact that cache misses are still a cause of significant performance loss with 2 MB and 4 MB caches. The question is, Why?

To better understand the answer to this question, we need to determine what factors contribute to the L3 miss rate and how they change as the L3 cache grows. Figure 4.12 shows this data, displaying the number of memory access cycles contributed per instruction from five sources. The two largest sources of L3 memory access cycles with a 1 MB L3 are instruction and capacity/conflict misses. With a larger L3 these two sources shrink to be minor contributors. Unfortunately, the compulsory, false sharing, and true sharing misses are unaffected by a larger L3. Thus, at 4 MB and 8 MB, the true sharing misses generate the dominant fraction of the misses; the lack of change in true sharing misses leads to the limited reductions in the overall miss rate when increasing the L3 cache size beyond 2 MB.

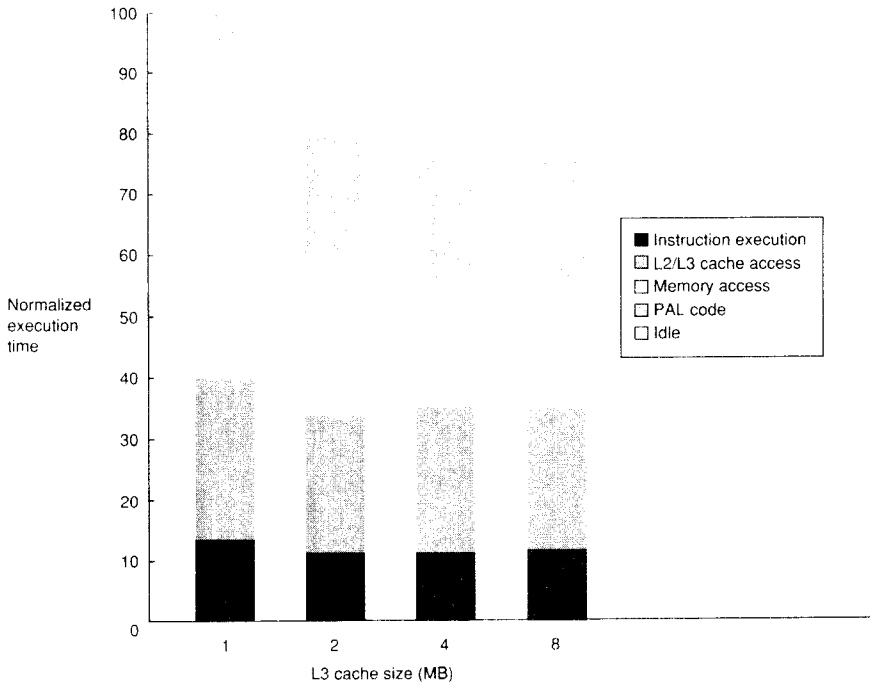

Figure 4.11 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 MB to 8 MB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system stalls, more server processes are needed to cover the I/O latency. The workload could be retuned to increase the computation/communication balance, holding the idle time in check

Increasing the cache size eliminates most of the uniprocessor misses, while leaving the multiprocessor misses untouched. How does increasing the processor count affect different types of misses? Figure 4.13 shows this data assuming a base configuration with a 2 MB, two-way set associative L3 cache. As we might expect, the increase in the true sharing miss rate, which is not compensated for by any decrease in the uniprocessor misses, leads to an overall increase in the memory access cycles per instruction.

The final question we examine is whether increasing the block size—which should decrease the instruction and cold miss rate and, within limits, also reduce the capacity/conflict miss rate and possibly the true sharing miss rate—is helpful for this workload. Figure 4.14 shows the number of misses per 1000 instructions as the block size is increased from 32 to 256. Increasing the block size from 32 to 256 affects four of the miss rate components:

- The true sharing miss rate decreases by more than a factor of 2, indicating some locality in the true sharing patterns.

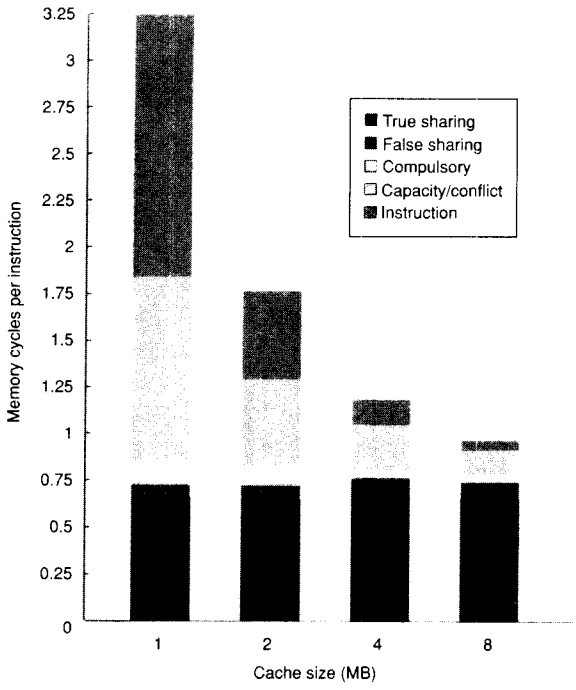

Figure 4.12 The contributing causes of memory access cycles shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

- The compulsory miss rate significantly decreases, as we would expect.
- The conflict/capacity misses show a small decrease (a factor of 1.26 compared to a factor of 8 increase in block size), indicating that the spatial locality is not high in the uniprocessor misses that occur with L3 caches larger than 2 MB.
- The false sharing miss rate, although small in absolute terms, nearly doubles.

The lack of a significant effect on the instruction miss rate is startling. If there were an instruction-only cache with this behavior, we would conclude that the spatial locality is very poor. In the case of a mixed L2 cache, other effects such as instruction-data conflicts may also contribute to the high instruction cache miss rate for larger blocks. Other studies have documented the low spatial locality in the instruction stream of large database and OLTP workloads, which have lots of short basic blocks and special-purpose code sequences. Nonetheless, increasing the block size of the third-level cache to 128 or possibly 256 bytes seems appropriate.

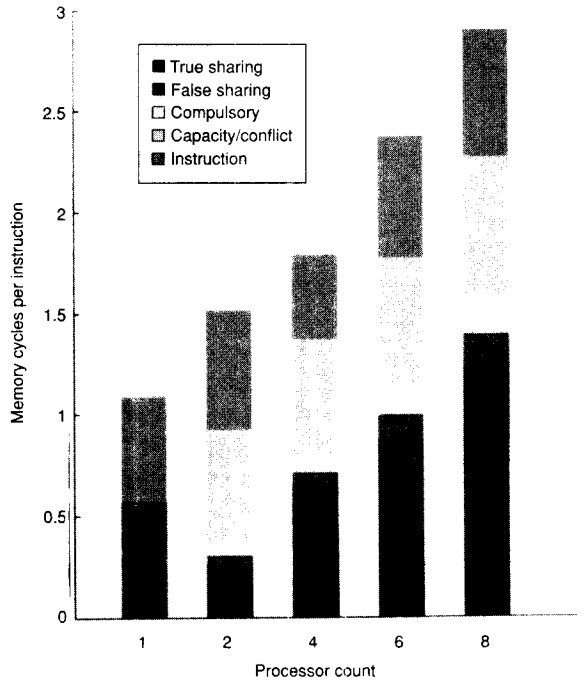

Figure 4.13 The contribution to memory access cycles increases as processor count increases primarily due to increased true sharing. The compulsory misses slightly increase since each processor must now handle more compulsory misses.

A Multiprogramming and OS Workload

Our next study is a multiprogrammed workload consisting of both user activity and OS activity. The workload used is two independent copies of the compile phases of the Andrew benchmark, a benchmark that emulates a software development environment. 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 processor is largely idle. The overall workload is much more system and I/O intensive than the highly tuned commercial workload.

For the workload measurements, we assume the following memory and I/O systems:

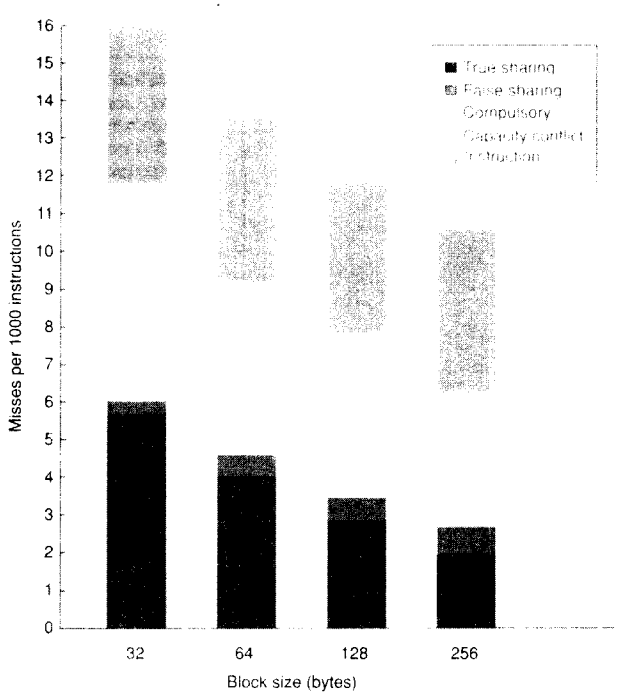

Figure 4.14 The number of misses per 1000 instructions drops steadily as the block size of the L3 cache is increased, making a good case for an L3 block size of at least 128 bytes. The L3 cache is 2 MB, two-way set associative.

- *Level 1 instruction cache*—32 KB, two-way set associative with a 64-byte block, 1 clock cycle hit time.
- *Level 1 data cache*—32 KB, two-way set associative with a 32-byte block, 1 clock cycle hit time. We vary the L1 data cache to examine its effect on cache behavior.
- *Level 2 cache*—1 MB 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 4.15 shows how the execution time breaks down for the eight processors using the parameters just listed. Execution time is broken into four components:

1. *Idle*—Execution in the kernel mode idle loop

	User execution	Kernel execution	Synchronization wait	CPU idle (waiting for I/O)
% instructions executed	27	3	1	69
% execution time	27	7	2	64

Figure 4.15 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 processors is active. These data and the subsequent measurements for this workload were collected with the SimOS system [Rosenblum et al. 1995]. The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University.

2. *User*—Execution in user code
3. *Synchronization*—Execution or waiting for synchronization variables
4. *Kernel*—Execution in the OS that is neither idle nor in synchronization access

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 64-byte block size, two-way 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. This partially accounts for the fact that although the user code executes nine times as many instructions as the kernel, those instructions take only about four times as long as the smaller number of instructions executed by the kernel.

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. 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.

Although the user code executes more instructions, the behavior of the operating system can cause more cache misses than the user processes for two reasons beyond larger code size and lack of locality. 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, and this component of the miss rate is small.

Figure 4.16 shows the data miss rate versus data cache size and versus block size for the kernel and user components. Increasing the data cache size affects the user miss rate more than it affects the kernel miss rate. Increasing the block size has beneficial effects for both miss rates, 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 rarer, the negative effects of increasing block size are small. To understand why the kernel and user processes behave differently, we can look at the how the kernel misses behave.

Figure 4.17 shows the variation in the kernel misses versus increases in cache size and in block size. The misses are broken into three classes: compulsory misses, coherence misses (from both true and false sharing), and capacity/conflict misses (which include misses caused by interference between the OS and the user process and between multiple user processes). Figure 4.17 confirms that, for the kernel references, increasing the cache size reduces solely the uniprocessor capacity/conflict miss rate. In contrast, increasing the block size causes a reduction in the compulsory miss rate. The absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are probably insignificant, although such misses may be offsetting some of the gains from reducing the true sharing misses.

If we examine the number of bytes needed per data reference, as in Figure 4.18, we see that the kernel has a higher traffic ratio that grows with block size. It

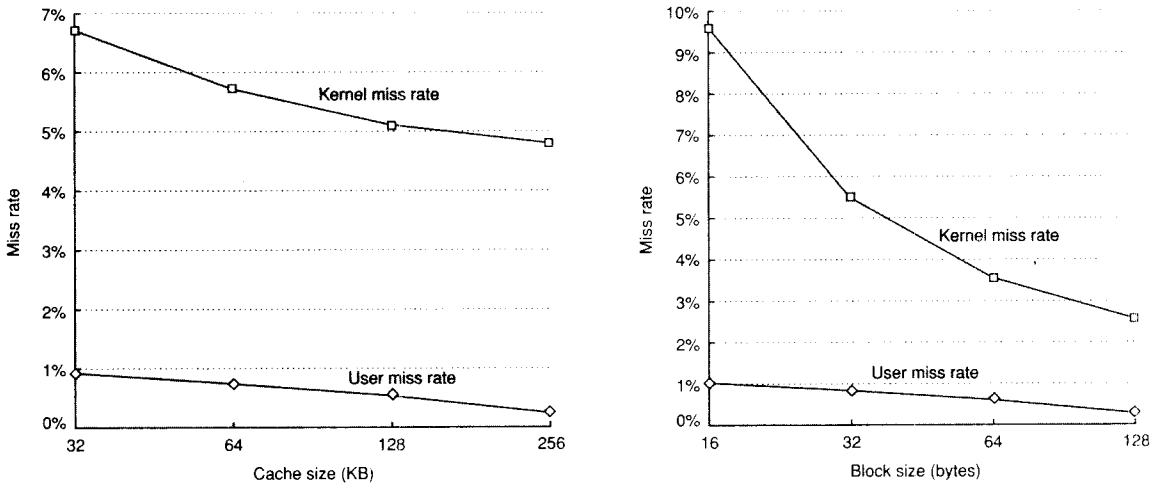

Figure 4.16 The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 KB to 256 KB (with a 32-byte block) causes the user miss rate to decrease proportionately more than the kernel miss rate: the user-level miss rate drops by almost a factor of 3, while the kernel-level miss rate drops only by a factor of 1.3. The miss rate for both user and kernel components drops steadily as the L1 block size is increased (while keeping the L1 cache at 32 KB). In contrast to the effects of increasing the cache size, increasing the block size improves the kernel miss rate more significantly (just under 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).

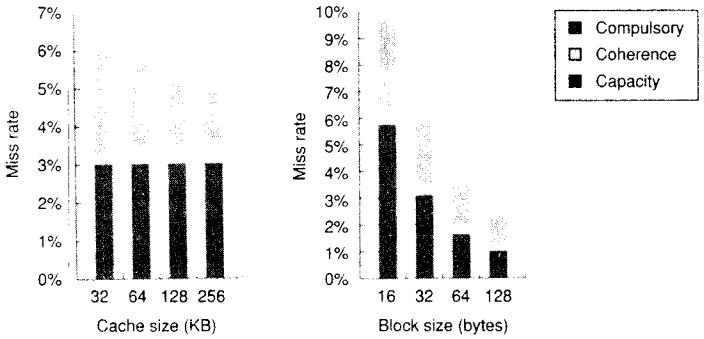

Figure 4.17 The components of the kernel data miss rate change as the L1 data cache size is increased from 32 KB to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant, since it is unaffected by cache size. The capacity component drops by more than a factor of 2, 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. As we would expect, the increasing block size of the L1 data cache 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%.

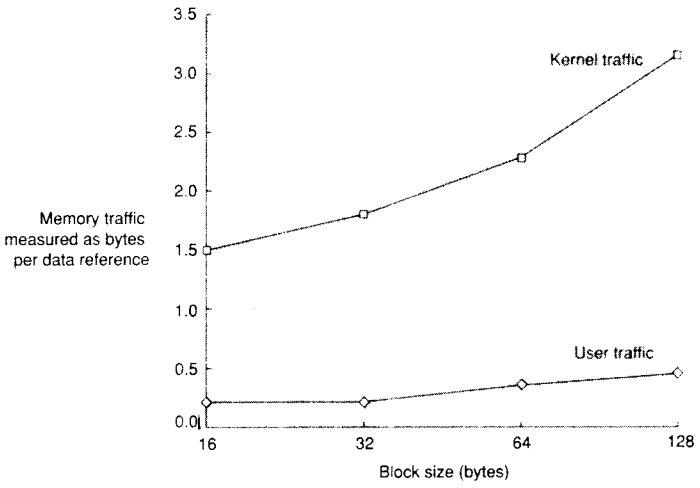

Figure 4.18 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart against the data on scientific programs shown in Appendix H.

is easy to see why this occurs: when going from a 16-byte block to a 128-byte block, the miss rate drops by about 3.7, but the number of bytes transferred per miss increases by 8, so the total miss traffic increases by just over a factor of 2. The user program also more than doubles as the block size goes from 16 to 128 bytes, but it starts out at a much lower level.

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, and the behavior is similar to what was measured for this workload, it will become very difficult to build a sufficiently capable memory system. One possible route to improving performance is to make the OS more cache aware, through either better programming environments or through programmer assistance. For example, the OS reuses memory for requests that arise from different system calls. Despite the fact that the reused memory will be completely overwritten, the hardware, not recognizing this, will attempt to preserve coherency and the possibility that some portion of a cache block may be read, even if it is not. This behavior is analogous to the reuse of stack locations on procedure invocations. The IBM Power series has support to allow the compiler to indicate this type of behavior on procedure invocations. It is harder to detect such behavior by the OS, and doing so may require programmer assistance, but the payoff is potentially even greater.

4.4 Distributed Shared Memory and Directory-Based Coherence

As we saw in Section 4.2, 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, a block size of 64 bytes, and a 512 KB data cache, the total bus bandwidth demand (ignoring stall cycles) for the four programs in the scientific/technical workload of Appendix H ranges from about 4 GB/sec to about 170 GB/sec, assuming a processor that sustains one data reference per clock, which for a 4 GHz clock is four data references per ns, which is what a 2006 superscalar processor with nonblocking caches might generate. In comparison, the memory bandwidth of the highest-performance centralized shared-memory 16-way multiprocessor in 2006 was 2.4 GB/sec per processor. In 2006, multiprocessors with a distributed-memory model are available with over 12 GB/sec per processor to the nearest memory.

We can increase the memory bandwidth and interconnection bandwidth by distributing the memory, as shown in Figure 4.2 on page 201; this immediately separates local memory traffic from remote memory traffic, reducing the bandwidth demands on the memory system and on the interconnection network. Unless we eliminate the need for the coherence protocol to broadcast on every cache miss, distributing the memory will gain us little.

As we mentioned earlier, the alternative to a snoop-based coherence 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. A directory protocol also can be used to reduce the bandwidth demands in a centralized shared-memory machine, as the Sun T1 design does (see Section 4.8.) We explain a directory protocol as if it were implemented with a distributed memory, but the same design also applies to a centralized memory organized into banks.

The simplest directory implementations associate an entry in the directory with each memory block. In such implementations, the amount of information is proportional to the product of the number of memory blocks (where each block is the same size as the level 2 or level 3 cache block) and the number of processors. This overhead is not a problem for multiprocessors with less than about 200 processors because the directory overhead with a reasonable block size will be tolerable. For larger multiprocessors, we need methods to allow the directory structure to be efficiently scaled. The methods that have been used 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 by using individual bits to stand for a small collection of processors.

To prevent the directory from becoming the bottleneck, the directory is distributed along with the memory (or with the interleaved memory banks in an SMP), so that different directory accesses can go to different directories, 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 4.19 shows how our distributed-memory multiprocessor 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 block that is currently shared 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.
- *Modified*—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.

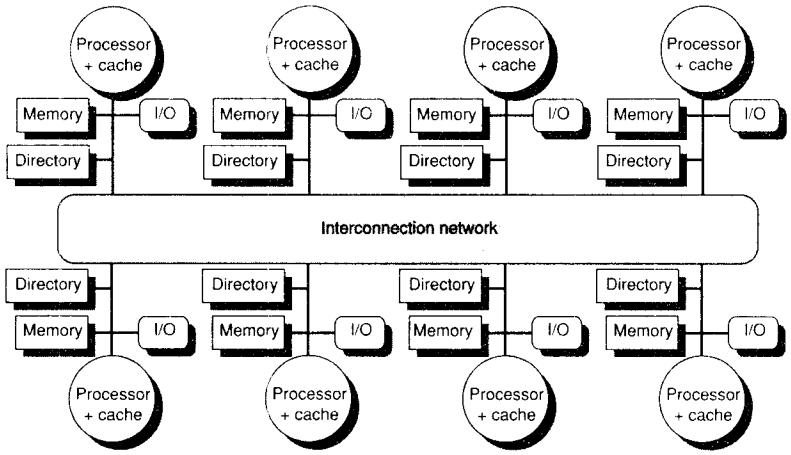

Figure 4.19 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over 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.

In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies 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. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

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 for the purpose of handling misses and maintaining coherence. Figure 4.20 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,

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	local cache	home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
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	D	Return a data value from the home memory.
Data write back	remote cache	home directory	A, D	Write back a data value for address A.

Figure 4.20 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting processor number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local cache to the home. The fourth through sixth 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.

while the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a *remote node.*

A remote node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

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 4.6 when we discuss memory consistency models. In this section, we use this assumption to ensure that invalidates sent by a processor are honored before new messages are transmitted, just as we assumed in the discussion of implementing snooping protocols. As we

did in the snooping case, we omit some details necessary to implement the coherence protocol. In particular, the serialization of writes and knowing that the invalidates for a write have completed are not as simple as in the broadcast-based snooping mechanism. Instead, explicit acknowledgements are required in response to write misses and invalidate requests. We discuss these issues in more detail in Appendix H.

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 multiprocessor (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 H.

Figure 4.21 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 4.21. An individual cache also generates read miss, write miss, and invalidate 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. Knowing when invalidates complete is a separate problem and is handled separately.

The operation of the state transition diagram for a cache block in Figure 4.21 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 (or other network) 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: updating the directory state and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a block; unlike in a snoopy scheme, however, the directory state indicates the state of 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 one node. In addition to the state of each block, the directory must track the set of processors

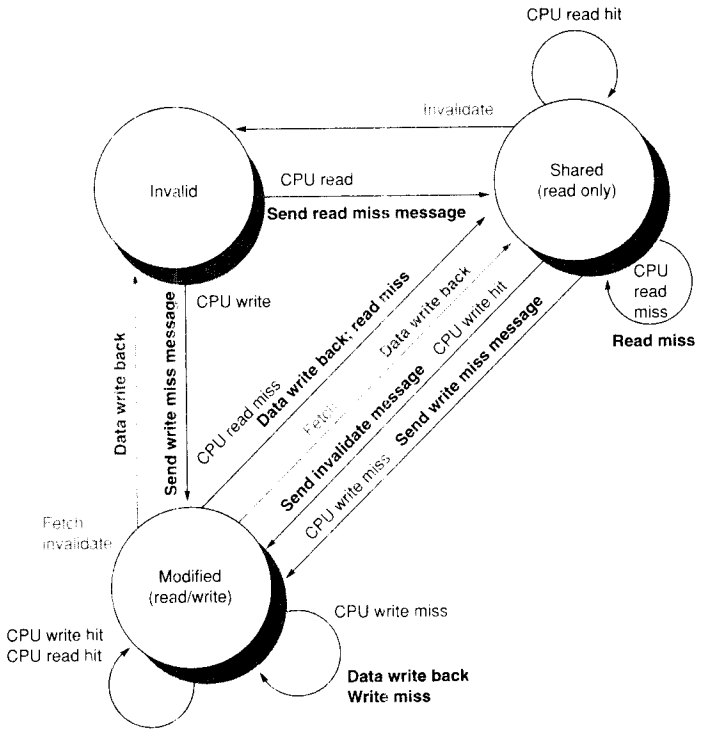

Figure 4.21 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in black, and those from the home directory are shown in gray. The states are identical to those in the snooping case, and the transactions are very similar, with 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.

that have a copy of a block; we use a set called *Sharers* to perform this function. In multiprocessors with less than 64 nodes (each of which may represent two to four times as many processors), this set is typically kept as a bit vector. In larger multiprocessors, other techniques are needed. Directory requests need to update the set Sharers and also read the set to perform invalidations.

Figure 4.22 shows the actions taken at the directory in response to messages received. The directory receives three different requests: read miss, write miss, 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.

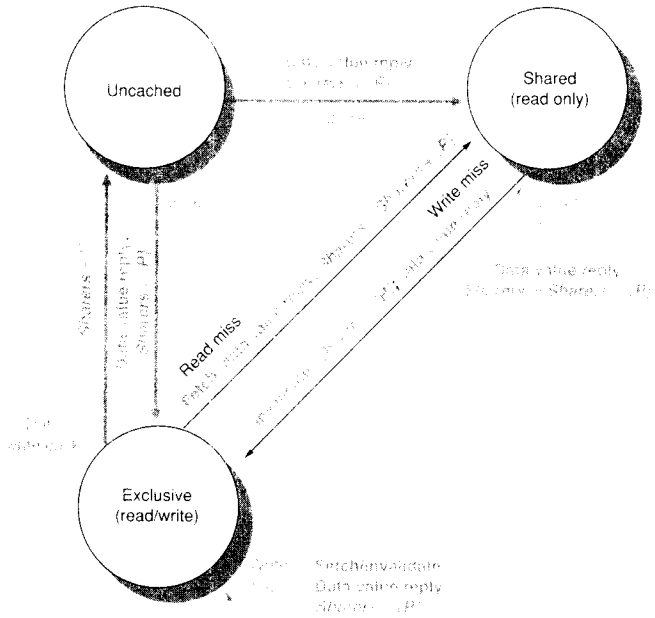

Figure 4.22 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request.

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 write back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value 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 4.22 is a simplification, just as it was in the snooping cache case. In the case of a directory, as well as a snooping scheme implemented with a network other than a bus, our protocols will need to deal with nonatomic memory transactions. Appendix H explores these issues in depth.

The directory protocols used in real multiprocessors contain additional optimizations. In particular, in this protocol 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 of the protocols in use in commercial multiprocessors forward the data from the owner node to the requesting node directly (as well as performing the write back to the home). Such optimizations often add complexity by increasing the possibility of deadlock and by increasing the types of messages that must be handled.

Implementing a directory scheme requires solving most of the same challenges we discussed for snoopy protocols beginning on page 217. There are, however, new and additional problems, which we describe in Appendix H.

4.5 Synchronization: The Basics

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. For smaller multiprocessors 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. In this section we focus on the implementation of lock and unlock synchronization operations. Lock and unlock can be used straightforwardly to create mutual exclusion, as well as to implement more complex synchronization mechanisms.

In larger-scale multiprocessors or high-contention situations, synchronization can become a performance bottleneck because contention introduces additional delays and because latency is potentially greater in such a multiprocessor. We discuss how the basic synchronization mechanisms of this section can be extended for large processor counts in Appendix H.

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 exchange*, 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 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 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 multiprocessors, 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.

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 requirement 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 is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively 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 1 if it was successful and a 0 otherwise. Since the load linked returns the initial value and the store conditional returns 1 only if it succeeds, 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
      DADDUI  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 until it succeeds. Spin locks are used when programmers expect the lock to be held for a very short amount of time and when they 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:

```
        DADDUI  R2,R0,#1
lockit: EXCH    R2,0(R1)     ;atomic exchange
        BNEZ    R2,lockit    ;already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks 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.

Obtaining 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:  LD      R2,0(R1)      ;load of lock
         BNEZ    R2,lockit     ;not available-spin
         DADDUI  R2,R0,#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 4.23 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 (P0)	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 (P2)	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1, and sets Lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; generates write back
8		Spins, testing if lock = 0			None

Figure 4.23 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 8 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 operations are explicitly separated. The load linked need not cause any bus traffic. This fact allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock, the LL has replaced the LD, and the SC has replaced the EXCH):

```
lockit:  LL      R2,0(R1)      ;load linked
         BNEZ    R2,lockit     ;not available-spin
         DADDUI  R2,R0,#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. We address this issue and other issues for larger processor counts in Appendix H.

4.6 Models of Memory Consistency: An Introduction

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 "how consistent" we mean, when must a processor see a value that has been updated by another processor? Since processors communicate through shared variables (used both for data values and 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 of how consistent memory must be seems simple, it is remarkably complicated, as we can see with a simple 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 memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. Sequential consistency eliminates the possibility of some nonobvious execution in the previous example because the assignments must be completed before the if statements are initiated.

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 effective 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 multiprocessor 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 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 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 coherence 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 for the write is 50 + 40 + 80 = 170 cycles. In comparison, the ownership time is only 50 cycles. With appropriate write buffer implementations, it is even possible to continue before ownership is established.

To provide better performance, researchers and architects have explored two different routes. First, they developed ambitious implementations that preserve sequential consistency but use latency-hiding techniques to reduce the penalty; we discuss these in Section 4.7. Second, they developed less restrictive memory consistency models that allow for faster hardware. Such models can affect how the programmer sees the multiprocessor, 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 are 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, which 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.

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 likely be unpredictable 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 multiprocessor. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the multiprocessor and the type of synchronization.

Finally, the use of standard synchronization primitives ensures that even if the architecture implements a more relaxed consistency model than sequential consistency, a synchronized program will behave as if the hardware implemented sequential consistency.

Relaxed Consistency Models: The Basics

The key idea in relaxed consistency models is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent. There are a variety of relaxed models that are classified according to what read and write orderings they relax. We specify the orderings by a set of rules of the form X→Y, meaning that operation X must complete before operation Y is done. Sequential consistency requires maintaining all four possible orderings: R→W, R→R, W→R, and W→W. The relaxed models are defined by which of these four sets of orderings they relax:

1. Relaxing the W→R ordering yields a model known as *total store ordering* or *processor consistency*. Because this ordering retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.

2. Relaxing the W→W ordering yields a model known as *partial store order*.
3. Relaxing the R→W and R→R orderings yields a variety of models including *weak ordering*, the PowerPC consistency model, and *release consistency*, depending on the details of the ordering restrictions and how synchronization operations enforce ordering.

By relaxing these orderings, the processor can possibly obtain significant performance advantages. There are, however, many complexities in describing relaxed consistency models, including the advantages and complexities of relaxing different orders, defining precisely what it means for a write to complete, and deciding when processors can see values that the processor itself has written. For more information about the complexities, implementation issues, and performance potential from relaxed models, we highly recommend the excellent tutorial by Adve and Gharachorloo [1996].

Final Remarks on Consistency Models

At the present time, many multiprocessors being built support some sort of relaxed consistency model, varying from processor consistency to release consistency. Since synchronization is highly multiprocessor 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.

An alternative viewpoint, which we discuss more extensively in the next section, argues that with speculation much of the performance advantage of relaxed consistency models can be obtained with sequential or processor consistency.

A key part of this argument in favor of relaxed consistency revolves around the role of the compiler and its ability to optimize memory access to potentially shared variables; this topic is also discussed in the next section.

4.7 Crosscutting Issues

Because multiprocessors redefine many system characteristics (e.g., performance assessment, memory latency, and the importance of scalability), they introduce interesting design problems that cut across the spectrum, affecting both hardware and software. In this section we give several examples related to the issue of memory consistency.

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 Speculation to Hide Latency in Strict Consistency Models

As we saw in Chapter 2, speculation can be used to hide memory latency. It can also be used to hide latency arising from a strict consistency model, giving much of the benefit of a relaxed memory model. The key idea is for the processor to use dynamic scheduling to reorder memory references, letting them possibly execute out of order. Executing the memory references out of order may generate violations of sequential consistency, which might affect the execution of the program. This possibility is avoided by using the delayed commit feature of a speculative processor. Assume the coherency protocol is based on invalidation. If the processor receives an invalidation for a memory reference before the memory reference is committed, the processor uses speculation recovery to back out the computation and restart with the memory reference whose address was invalidated.

If the reordering of memory requests by the processor yields an execution order that could result in an outcome that differs from what would have been seen under sequential consistency, the processor will redo the execution. The key to using this approach is that the processor need only guarantee that the result would be the same as if all accesses were completed in order, and it can achieve this by detecting when the results might differ. The approach is attractive because the speculative restart will rarely be triggered. It will only be triggered when there are unsynchronized accesses that actually cause a race [Gharachorloo, Gupta, and Hennessy 1992].

Hill [1998] advocates the combination of sequential or processor consistency together with speculative execution as the consistency model of choice. His argument has three parts. First, an aggressive implementation of either sequential consistency or processor consistency will gain most of the advantage of a more relaxed model. Second, such an implementation adds very little to the implementation cost of a speculative processor. Third, such an approach allows the programmer to reason using the simpler programming models of either sequential or processor consistency.

The MIPS R10000 design team had this insight in the mid-1990s and used the R10000's out-of-order capability to support this type of aggressive implementation of sequential consistency. Hill's arguments are likely to motivate others to follow this approach.

One open question is how successful compiler technology will be in optimizing memory references to shared variables. The state of optimization technology and the fact that shared data are often accessed via pointers or array indexing

have limited the use of such optimizations. If this technology became available and led to significant performance advantages, compiler writers would want to be able to take advantage of a more relaxed programming model.

Inclusion and Its Implementation

All 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 that occurs when snoops and processor cache accesses must contend for the cache. Many multiprocessors with multilevel caches enforce the inclusion property, although recent multiprocessors with smaller L1 caches and different block sizes have sometimes chosen not to enforce inclusion. 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 sizes 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 two 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; then 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; different levels of associativity create the same sort of problems. In 2006, designers appear to be split on the enforcement of inclusion. Baer and Wang [1988] describe the advantages and challenges of inclusion in detail.

4.8 Putting It All Together: The Sun T1 Multiprocessor

T1 is a multicore multiprocessor introduced by Sun in 2005 as a server processor. What makes T1 especially interesting is that it is almost totally focused on exploiting thread-level parallelism (TLP) rather than instruction-level parallelism (ILP). Indeed, it is the only single-issue desktop or server microprocessor introduced in more than five years. Instead of focusing on ILP, T1 puts all its attention on TLP, using both multiple cores and multithreading to produce throughput.

Each T1 processor contains eight processor cores, each supporting four threads. Each processor core consists of a simple six-stage, single-issue pipeline (a standard five-stage RISC pipeline like that of Appendix A, with one stage added for thread switching). T1 uses fine-grained multithreading, switching to a new thread on each clock cycle, and threads that are idle because they are waiting due to a pipeline delay or cache miss are bypassed in the scheduling. The processor is idle only when all four threads are idle or stalled. Both loads and branches incur a 3-cycle delay that can only be hidden by other threads. A single set of floating-point functional units is shared by all eight cores, as floating-point performance was not a focus for T1.

Figure 4.24 shows the organization of the T1 processor. The cores access four level 2 caches via a crossbar switch, which also provides access to the shared floating-point unit. Coherency is enforced among the L1 caches by a directory associated with each L2 cache block. The directory operates analogously to those we discussed in Section 4.4, but is used to track which L1 caches have copies of an L2 block. By associating each L2 cache with a particular memory bank and enforcing the subset property, T1 can place the directory at L2 rather than at the memory, which reduces the directory overhead. Because the L1 data cache is write through, only invalidation messages are required; the data can always be retrieved from the L2 cache.

Figure 4.25 summarizes the T1 processor.

T1 Performance

We look at the performance of T1 using three server-oriented benchmarks: TPC-C, SPECJBB (the SPEC Java Business Benchmark), and SPECWeb99. The SPECWeb99 benchmark is run on a four-core version of T1 because it cannot scale to use the full 32 threads of an eight-core processor; the other two benchmarks are run with eight cores and 4 threads each for a total of 32 threads.

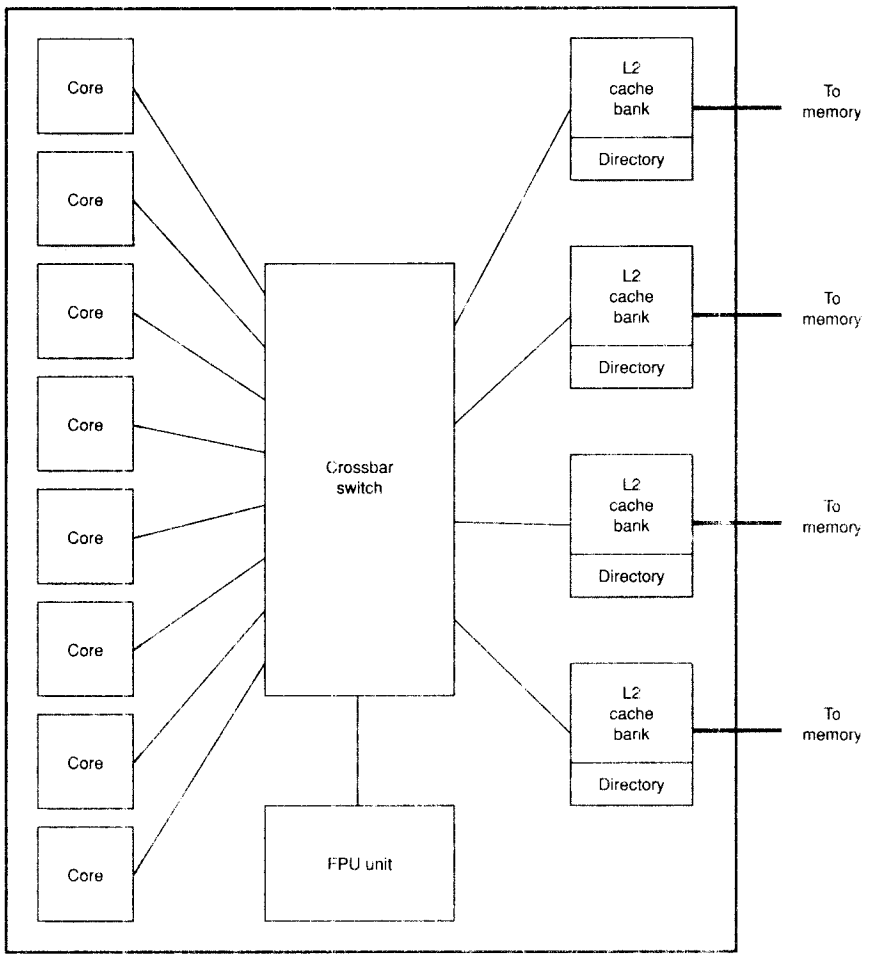

Figure 4.24 The T1 processor. Each core supports four threads and has its own level 1 caches (16 KB for instructions and 8 KB for data). The level 2 caches total 3 MB and are effectively 12-way associative. The caches are interleaved by 64-byte cache lines.

We begin by looking at the effect of multithreading on the performance of the memory system when running in single-threaded versus multithreaded mode. Figure 4.26 shows the relative increase in the miss rate and the observed miss latency when executing with 1 thread per core versus executing 4 threads per core for TPC-C. Both the miss rates and the miss latencies increase, due to increased contention in the memory system. The relatively small increase in miss latency indicates that the memory system still has unused capacity.

As we demonstrated in the previous section, the performance of multiprocessor workloads depends intimately on the memory system and the interaction with

Characteristic	Sun T1
Multiprocessor and multithreading support	Eight cores per chip; four threads per core. Fine-grained thread scheduling. One shared floating-point unit for eight cores. Supports only on-chip multiprocessing.
Pipeline structure	Simple, in-order, six-deep pipeline with 3-cycle delays for loads and branches.
L1 caches	16 KB instructions; 8 KB data. 64-byte block size. Miss to L2 is 23 cycles, assuming no contention.
L2 caches	Four separate L2 caches, each 750 KB and associated with a memory bank. 64-byte block size. Miss to main memory is 110 clock cycles assuming no contention.
Initial implementation	90 nm process; maximum clock rate of 1.2 GHz; power 79 W; 300M transistors, 379 mm^2 die.

Figure 4.25 A summary of the T1 processor.

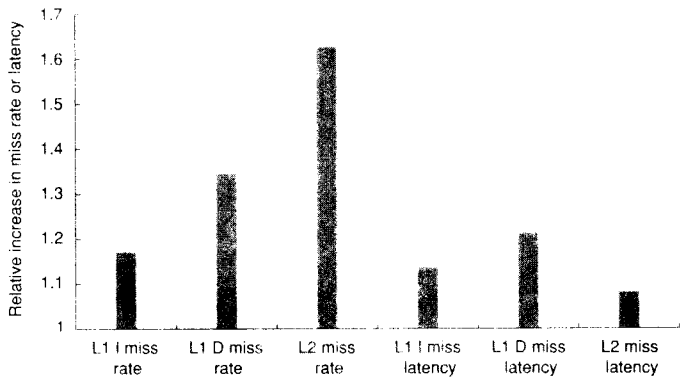

Figure 4.26 The relative change in the miss rates and miss latencies when executing with 1 thread per core versus 4 threads per core on the TPC-C benchmark. The latencies are the actual time to return the requested data after a miss. In the 4-thread case, the execution of other threads could potentially hide much of this latency.

the application. For T1 both the L2 cache size and the block size are key parameters. Figure 4.27 shows the effect on miss rates from varying the L2 cache size by a factor of 2 from the base of 3 MB and by reducing the block size to 32 bytes. The data clearly show a significant advantage of a 3 MB L2 versus a 1.5 MB; further improvements can be gained from a 6 MB L2. As we can see, the choice of a 64-byte block size reduces the miss rate but by considerably less than a factor of 2. Hence, using the larger block size T1 generates more traffic to the memories. Whether this has a significant performance impact depends on the characteristics of the memory system.

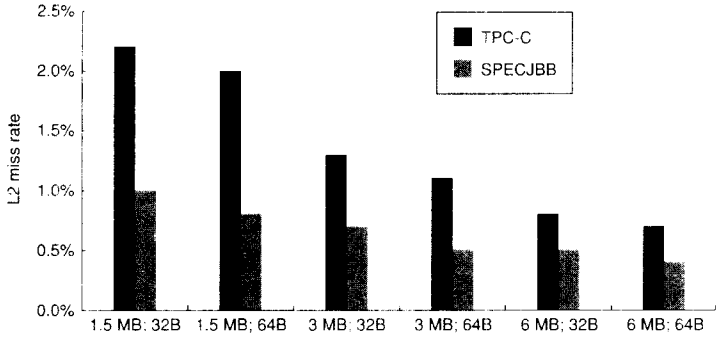

Figure 4.27 Change in the L2 miss rate with variation in cache size and block si: Both TPC-C and SPECJBB are run with all eight cores and four threads per core. Rec that T1 has a 3 MB L2 with 64-byte lines.

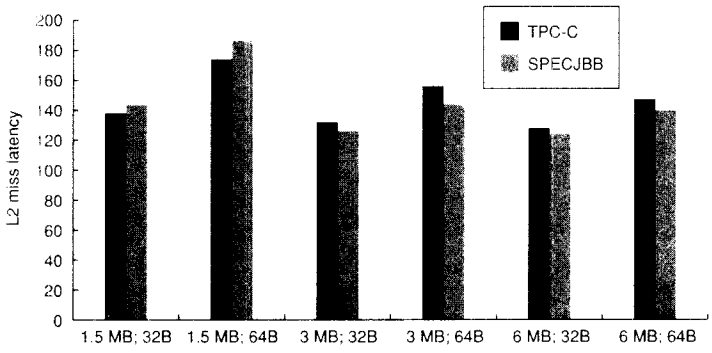

Figure 4.28 The change in the miss latency of the L2 cache as the cache size and block size are changed. Although TPC-C has a significantly higher miss rate, its miss penalty is only slightly higher. This is because SPECJBB has a much higher dirty miss rate, requiring L2 cache lines to be written back with high frequency. Recall that T1 has a 3 MB L2 with 64-byte lines.

As we mentioned earlier, there is some contention at the memory from multiple threads. How do the cache size and block size affect the contention at the memory system? Figure 4.28 shows the effect on the L2 cache miss latency under the same variations as we saw in Figure 4.27. As we can see, for either a 3 MB or 6 MB cache, the larger block size results in a smaller L2 cache miss time. How can this be if the miss rate changes much less than a factor of 2? As we will see in more detail in the next chapter, modern DRAMs provide a block of data for only slightly more time than needed to provide a single word; thus, the miss penalty for the 32-byte block is only slightly less than the 64-byte block.

Overall Performance

Figure 4.29 shows the per-thread and per-core CPI, as well as the effective instructions per clock (IPC) for the eight-processor chip. Because T1 is a fine-grained multithreaded processor with four threads per core, with sufficient parallelism the ideal effective CPI per thread would be 4, since that would mean that each thread was consuming one cycle out of every four. The ideal CPI per core would be 1. The effective IPC for T1 is simply 8 divided by the per-core CPI.

At first glance, one might react that T1 is not very efficient, since the effective throughout is between 56% and 71% of the ideal on these three benchmarks. But, consider the comparative performance of a wide-issue superscalar. Processors such as the Itanium 2 (higher transistor count, much higher power, comparable silicon area) would need to achieve incredible instruction throughput sustaining 4.5–5.7 instructions per clock, well more than double the acknowledged IPC. It appears quite clear that, at least for integer-oriented server applications with thread-level parallelism, a multicore approach is a much better alternative than a single very wide issue processor. The next subsection offers some performance comparisons among multicore processors.

By looking at the behavior of an average thread, we can understand the interaction between multithreading and parallel processing. Figure 4.30 shows the percentage of cycles for which a thread is executing, ready but not executing, and not ready. Remember that not ready does not imply that the core with that thread is stalled; it is only when all four threads are not ready that the core will stall.

Threads can be not ready due to cache misses, pipeline delays (arising from long latency instructions such as branches, loads, floating point, or integer multiply/divide), and a variety of smaller effects. Figure 4.31 shows the relative frequency of these various causes. Cache effects are responsible for the thread not being ready from 50% to 75% of the time, with L1 instruction misses, L1 data misses, and L2 misses contributing roughly equally. Potential delays from the pipeline (called "pipeline delay") are most severe in SPECJBB and may arise from its higher branch frequency.

Benchmark	Per-thread CPI	Per core CPI	Effective CPI for eight cores	Effective IPC for eight cores
TPC-C	7.2	1.8	0.225	4.4
SPECJBB	5.6	1.40	0.175	5.7
SPECWeb99	6.6	1.65	0.206	4.8

Figure 4.29 The per-thread CPI, the per-core CPI, the effective eight-core CPI, and the effective IPC (inverse of CPI) for the eight-core T1 processor.

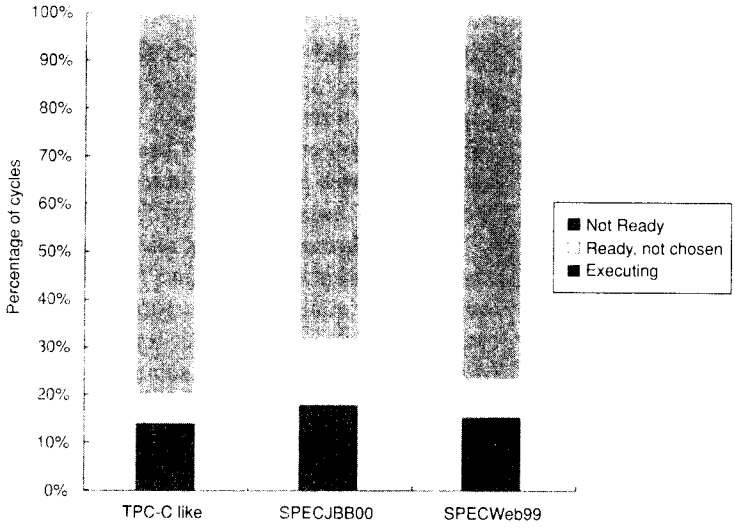

Figure 4.30 Breakdown of the status on an average thread. Executing indicates the thread issues an instruction in that cycle. Ready but not chosen means it could issue, but another thread has been chosen, and *not ready* indicates that the thread is awaiting the completion of an event (a pipeline delay or cache miss, for example).

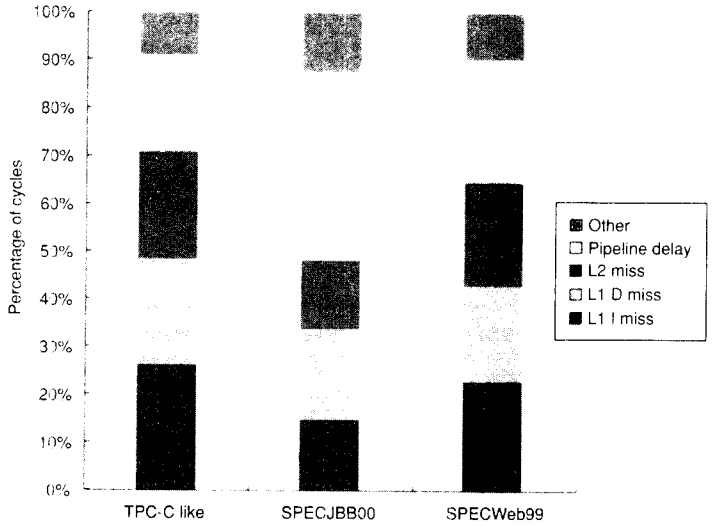

Figure 4.31 The breakdown of causes for a thread being not ready. The contribution to the "other" category varies. In TPC-C, store buffer full is the largest contributor; in SPEC-JBB, atomic instructions are the largest contributor; and in SPECWeb99, both factors contribute.

Performance of Multicore Processors on SPEC Benchmarks

Among recent processors, T1 is uniquely characterized by an intense focus on thread-level parallelism versus instruction-level parallelism. It uses multithreading to achieve performance from a simple RISC pipeline, and it uses multiprocessing with eight cores on a die to achieve high throughput for server applications. In contrast, the dual-core Power5, Opteron, and Pentium D use both multiple issue and multicore. Of course, exploiting significant ILP requires much bigger processors, with the result being that fewer cores fit on a chip in comparison to T1. Figure 4.32 summarizes the features of these multicore chips.

In addition to the differences in emphasis on ILP versus TLP, there are several other fundamental differences in the designs. Among the most important are

- There are significant differences in floating-point support and performance. The Power5 puts a major emphasis on floating-point performance, the Opteron and Pentium allocate significant resources, and the T1 almost ignores it. As a result, Sun is unlikely to provide any benchmark results for floating-point applications. A comparison that included only integer programs would be unfair to the three processors that include significant floating-point hardware (and the silicon and power cost associated with it). In contrast, a comparison using only floating-point applications would be unfair to the T1.
- The multiprocessor expandability of these systems differs and that affects the memory system design and the use of external interfaces. Power5 is designed

Characteristic	SUN T1	AMD Opteron	Intel Pentium D	IBM Power5
Cores	8	2	2	2
Instruction issues per clock per core	1	3	3	4
Multithreading	Fine-grained	No	SMT	SMT
Caches L1 I/D in KB per core L2 per core/shared L3 (off-chip)	16/8 3 MB shared	64/64 1 MB/core	12K uops/16 1 MB/core	64/32 L2: 1.9 MB shared L3: 36 MB
Peak memory bandwidth (DDR2 DRAMs)	34.4 GB/sec	8.6 GB/sec	4.3 GB/sec	17.2 GB/sec
Peak MIPS FLOPS	9600 1200	7200 4800 (w. SSE)	9600 6400 (w. SSE)	7600 7600
Clock rate (GHz)	1.2	2.4	3.2	1.9
Transistor count (M)	300	233	230	276
Die size (mm^2)	379	199	206	389
Power (W)	79	110	130	125

Figure 4.32 Summary of the features and characteristics of four multicore processors.

for the most expandability. The Pentium and Opteron design offer limited multiprocessor support. The T1 is not expandable to a larger system.

- The implementation technologies vary, making comparisons based on die size and power more difficult.
- There are significant differences in the assumptions about memory systems and the memory bandwidth available. For benchmarks with high cache miss rates, such as TPC-C and similar programs, the processors with larger memory bandwidth have a significant advantage.

Nonetheless, given the importance of the trade-off between ILP-centric and TLP-centric designs, it would be useful to try to quantify the performance differences as well as the efficacy of the approaches. Figure 4.33 shows the performance of the four multicore processors using the SPECRate CPU benchmarks, the SPECJBB2005 Java business benchmark, the SPECWeb05 Web server benchmark, and a TPC-C-like benchmark.

Figure 4.34 shows efficiency measures in terms of performance per unit die area and per watt for the four dual-core processors, with the results normalized to the measurement on the Pentium D. The most obvious distinction is the significant advantage in terms of performance/watt for the Sun T1 processor on the TPC-C-like and SPECJBB05 benchmarks. These measurements clearly demonstrate that for multithreaded applications, a TLP approach may be much more power efficient than an ILP-intensive approach. This is the strongest evidence to date that the TLP route may provide a way to increase performance in a power-efficient fashion.

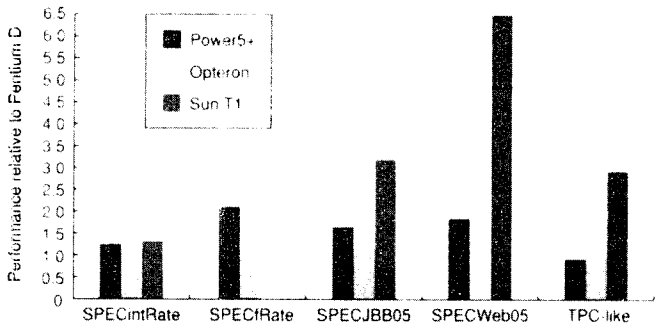

Figure 4.33 Four dual-core processors showing their performance on a variety of SPEC benchmarks and a TPC-C-like benchmark. All the numbers are normalized to the Pentium D (which is therefore at 1 for all the benchmarks). Some results are estimates from slightly larger configurations (e.g., four cores and two processors, rather than two cores and one processor), including the Opteron SPECJBB2005 result, the Power5 SPECWeb05 result, and the TPC-C results for the Power5, Opteron, and Pentium D. At the current time, Sun has refused to release SPECRate results for either the integer or FP portion of the suite.

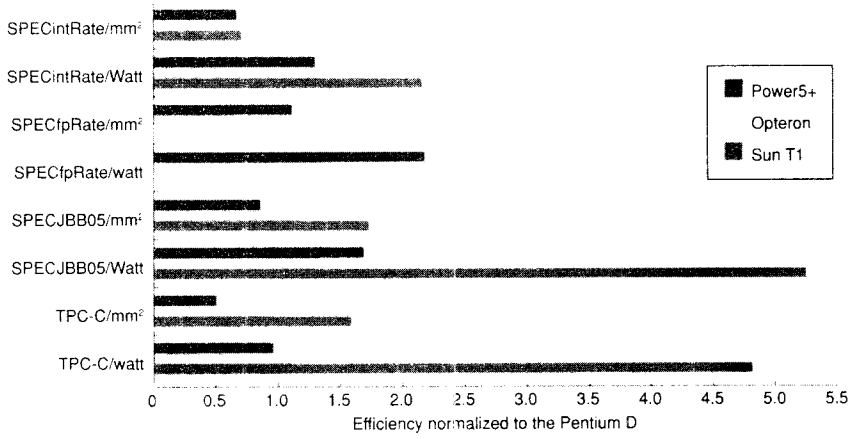

Figure 4.34 Performance efficiency on SPECRate for four dual-core processors, normalized to the Pentium D metric (which is always 1).

It is too early to conclude whether the TLP-intensive approaches will win across the board. If typical server applications have enough threads to keep T1 busy and the per-thread performance is acceptable, the T1 approach will be tough to beat. If single-threaded performance remains important in server or desktop environments, then we may see the market further fracture with significantly different processors for throughput-oriented environments and environments where higher single-thread performance remains important.

4.9 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 has surrounded 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 486s may be slower than the sequential version on a Pentium 4. 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 computer. Comparing the identical code on two computers 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, some scientific applications regularly achieve superlinear speedup for small increases in processor count (2 or 4 to 8 or 16). These results usually arise because critical data structures that do not fit into the aggregate caches of a multiprocessor with 2 or 4 processors fit into the aggregate cache of a multiprocessor with 8 or 16 processors.

In summary, comparing performance by comparing speedups is at best tricky and at worst misleading. Comparing the speedups for two different multiprocessors does not necessarily tell us anything about the relative performance of the multiprocessors. Even comparing two different algorithms on the same multiprocessor 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 1.9) had been broken by an MIMD multiprocessor. This statement 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. Because the running time grew faster than linear, the program actually ran longer after scaling, even with 1000 processors.

Speedup that assumes scaling of the input is not the same as true speedup and reporting it as if it were misleading. Since parallel benchmarks are often run on different-sized multiprocessors, it is important to specify what type of application scaling is permissible and how that scaling should be done. Although simply scaling the data size with processor count is rarely appropriate, assuming a fixed problem size for a much larger processor count is often inappropriate as well, since it is likely that users given a much larger multiprocessor would opt to run a larger or more detailed version of an application. In Appendix H, we discuss different methods for scaling applications for large-scale multiprocessors, introducing a model called *time-constrained scaling*, which scales the application data size so that execution time remains constant across a range of processor counts.

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 multiprocessor 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, I/O, and the overhead of the system (box, power supply, interconnect, etc.).

The effect of including memory in the system cost was pointed out by Wood and Hill [1995]. We use an example based on more recent data using TPC-C and SPECRate benchmarks, but the argument could also be made with a parallel scientific application workload, which would likely make the case even stronger.

Figure 4.35 shows the speedup for TPC-C, SPECintRate and SPECfpRate on an IBM eserver p5 multiprocessor configured with 4 to 64 processors. The figure shows that only TPC-C achieves better than linear speedup. For SPECintRate and SPECfpRate, speedup is less than linear, but so is the cost, since unlike TPC-C the amount of main memory and disk required both scale less than linearly.

As Figure 4.36 shows, larger processor counts can actually be more cost-effective than the four-processor configuration. In the future, as the cost of multiple processors decreases compared to the cost of the support infrastructure (cabinets, power supplies, fans, etc.), the performance/cost ratio of larger processor configurations will improve further.

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.

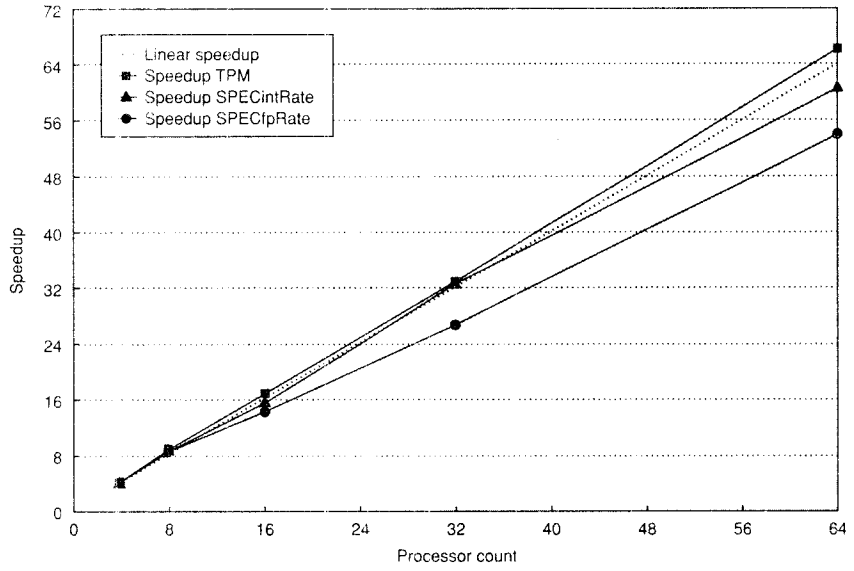

Figure 4.35 Speedup for three benchmarks on an IBM eserver p5 multiprocessor when configured with 4, 8, 16, 32, and 64 processors. The dashed line shows linear speedup.

Fallacy *Scalability is almost free.*

The goal of scalable parallel computing was a focus of much of the research and a significant segment of the high-end multiprocessor development from the mid-1980s through the late 1990s. In the first half of that period, it was widely held that you could build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small to large number of processors without sacrificing cost-effectiveness. The difficulty with this view is that multiprocessors 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 operating system support, reliability, and reconfigurability.

As an example, consider the Cray T3E, which used a 3D torus capable of scaling to 2048 processors as an interconnection network. At 128 processors, it delivers a peak bisection bandwidth of 38.4 GB/sec, or 300 MB/sec per processor. But for smaller configurations, the contemporaneous Compaq AlphaServer ES40 could accept up to 4 processors and has 5.6 GB/sec of interconnect bandwidth, or almost four times the bandwidth per processor. Furthermore, the cost per processor in a Cray T3E is several times higher than the cost in the ES40.

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

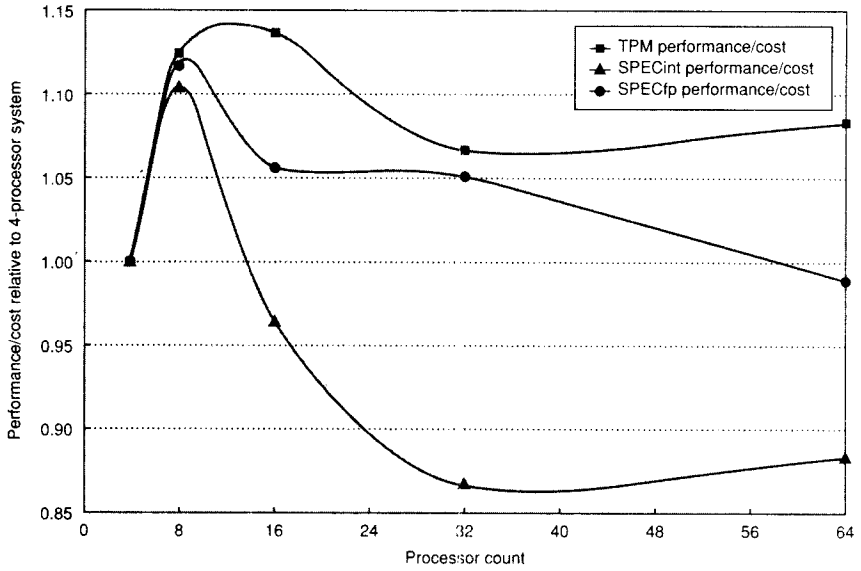

Figure 4.36 The performance/cost relative to a 4-processor system for three benchmarks run on an IBM eserver p5 multiprocessor containing from 4 to 64 processors shows that the larger processor counts can be as cost-effective as the 4-processor configuration. For TPC-C the configurations are those used in the official runs, which means that disk and memory scale nearly linearly with processor count, and a 64-processor machine is approximately twice as expensive as a 32-processor version. In contrast, the disk and memory are scaled more slowly (although still faster than necessary to achieve the best SPECRate at 64 processors). In particular the disk configurations go from one drive for the 4-processor version to four drives (140 GB) for the 64-processor version. Memory is scaled from 8 GB for the 4-processor system to 20 GB for the 64-processor system.

program. Obtaining scalability for real applications, as opposed to toys or small kernels, across factors of more than five in processor count, is a *major* challenge. In the future, new programming approaches, better compiler technology, and performance analysis tools may help with this critical problem, on which little progress has been made in 30 years.

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 processors, possibly because the software problems are much harder. We give one example to show the subtlety of the issues, but there are many examples we could choose from!

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the SGI

operating system originally protected 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, 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. Similar problems exist in memory structures, which increases the coherence traffic in cases where no sharing is actually occurring.

4.10 Concluding Remarks

For more than 30 years, researchers and designers have predicted the end of uniprocessors and their dominance by multiprocessors. During this time period the rise of microprocessors and their rapid performance growth has largely limited the role of multiprocessing to limited market segments. In 2006, we are clearly at an inflection point where multiprocessors and thread-level parallelism will play a greater role across the entire computing spectrum. This change is driven by several phenomena:

1. The use of parallel processing in some domains is much better understood. 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, as we discuss further in Appendix H.
2. The growth in server applications for transaction processing and Web services, as well as multiprogrammed environments, has been enormous, and

these applications have inherent and more easily exploited parallelism, through the processing of independent threads

3. After almost 20 years of breakneck performance improvement, we are in the region of diminishing returns for exploiting ILP, at least as we have known it. Power issues, complexity, and increasing inefficiency has forced designers to consider alternative approaches. Exploiting thread-level parallelism is the next natural step.
4. Likewise, for the past 50 years, improvements in clock rate have come from improved transistor speed. As we begin to see reductions in such improvements both from technology limitations and from power consumption, exploiting multiprocessor parallelism is increasingly attractive.

In the 1995 edition of this text, we concluded the chapter with a discussion of two then-current controversial issues:

1. What architecture would very large-scale, microprocessor-based multiprocessors use?
2. What was the role for multiprocessing in the future of microprocessor architecture?

The intervening years have largely resolved these two questions.

Because very large-scale multiprocessors did not become a major and growing market, the only cost-effective way to build such large-scale multiprocessors was to use clusters where the individual nodes are either single microprocessors or moderate-scale, shared-memory multiprocessors, which are simply incorporated into the design. We discuss the design of clusters and their interconnection in Appendices E and H.

The answer to the second question has become clear only recently, but it has become astonishingly clear. The future performance growth in microprocessors, at least for the next five years, will almost certainly come from the exploitation of thread-level parallelism through multicore processors rather than through exploiting more ILP. In fact, we are even seeing designers opt to exploit less ILP in future processors, instead concentrating their attention and hardware resources on more thread-level parallelism. The Sun T1 is a step in this direction, and in March 2006, Intel announced that its next round of multicore processors would be based on a core that is less aggressive in exploiting ILP than the Pentium 4 Netburst core. The best balance between ILP and TLP will probably depend on a variety of factors including the applications mix.

In the 1980s and 1990s, with the birth and development of ILP, software in the form of optimizing compilers that could exploit ILP was key to its success. Similarly, the successful exploitation of thread-level parallelism will depend as much on the development of suitable software systems as it will on the contributions of computer architects. Given the slow progress on parallel software in the past thirty-plus years, it is likely that exploiting thread-level parallelism broadly will remain challenging for years to come.

4.11 Historical Perspective and References

Section K.5 on the companion CD looks at the history of multiprocessors and parallel processing. Divided by both time period and architecture, the section includes discussions on early experimental multiprocessors and some of the great debates in parallel processing. Recent advances are also covered. References for further reading are included.

Case Studies with Exercises by David A. Wood

Case Study 1: Simple, Bus-Based Multiprocessor

Concepts illustrated by this case study

- Snooping Coherence Protocol Transitions
- Coherence Protocol Performance
- Coherence Protocol Optimizations
- Synchronization

The simple, bus-based multiprocessor illustrated in Figure 4.37 represents a commonly implemented symmetric shared-memory architecture. Each processor has a single, private cache with coherence maintained using the snooping coherence protocol of Figure 4.7. Each cache is direct-mapped, with four blocks each holding two words. To simplify the illustration, the cache-address tag contains the full address and each word shows only two hex characters, with the least significant word on the right. The coherence states are denoted M, S, and I for Modified, Shared, and Invalid.

4.1 [10/10/10/10/10/10/10] <4.2> For each part of this exercise, assume the initial cache and memory state as illustrated in Figure 4.37. Each part of this exercise specifies a sequence of one or more CPU operations of the form:

```
P#: <op> <address> [ <-- <value> ]
```

where P# designates the CPU (e.g., P0), <op> is the CPU operation (e.g., read or write), <address> denotes the memory address, and <value> indicates the new word to be assigned on a write operation.

Treat each action below as independently applied to the initial state as given in Figure 4.37. What is the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the given action? Show only the blocks that change, for example, P0.B0: (I, 120, 00 01) indicates that CPU P0's block B0 has the final state of I, tag of 120, and data words 00 and 01. Also, what value is returned by each read operation?

a. [10] <4.2> `P0: read 120`

b. [10] <4.2> `P0: write 120 <-- 80`

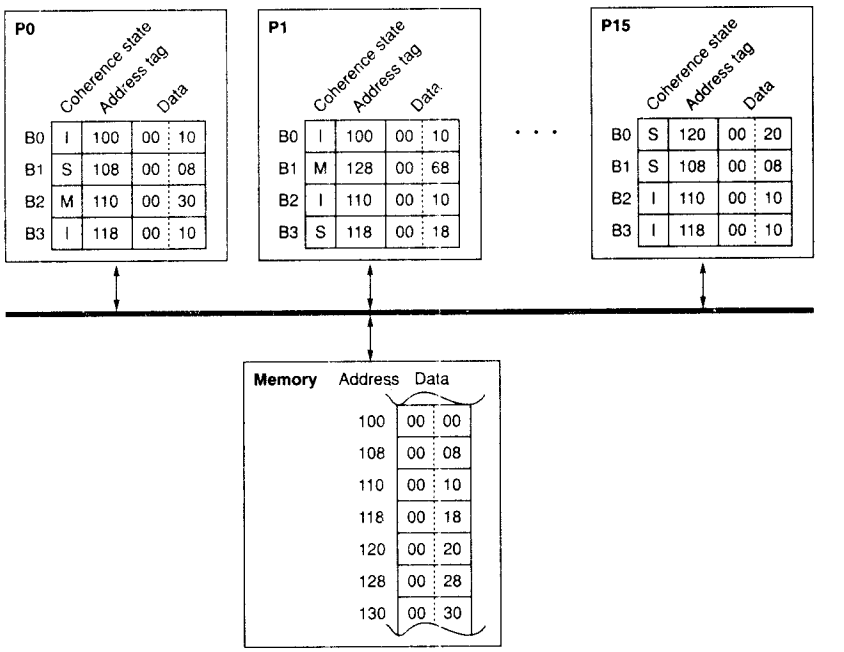

Figure 4.37 Bus-based snooping multiprocessor.

c. [10] <4.2> P15: write 120 <-- 80

d. [10] <4.2> P1: read 110

e. [10] <4.2> P0: write 108 <-- 48

f. [10] <4.2> P0: write 130 <-- 78

g. [10] <4.2> P15: write 130 <-- 78

4.2 [20/20/20/20] <4.3> The performance of a snooping cache-coherent multiprocessor depends on many detailed implementation issues that determine how quickly a cache responds with data in an exclusive or M state block. In some implementations, a CPU read miss to a cache block that is exclusive in another processor's cache is faster than a miss to a block in memory. This is because caches are smaller, and thus faster, than main memory. Conversely, in some implementations, misses satisfied by memory are faster than those satisfied by caches. This is because caches are generally optimized for "front side" or CPU references, rather than "back side" or snooping accesses.

For the multiprocessor illustrated in Figure 4.37, consider the execution of a sequence of operations on a single CPU where

- CPU read and write hits generate no stall cycles.
- CPU read and write misses generate N_{memory} and N_{cache} stall cycles if satisfied by memory and cache, respectively.

- CPU write hits that generate an invalidate incur $N_{invalidate}$ stall cycles.
- a writeback of a block, either due to a conflict or another processor's request to an exclusive block, incurs an additional $N_{writeback}$ stall cycles.

Consider two implementations with different performance characteristics summarized in Figure 4.38.

Consider the following sequence of operations assuming the initial cache state in Figure 4.37. For simplicity, assume that the second operation begins after the first completes (even though they are on different processors):

```
P1: read 110
P15: read 110
```

For Implementation 1, the first read generates 80 stall cycles because the read is satisfied by P0's cache. P1 stalls for 70 cycles while it waits for the block, and P0 stalls for 10 cycles while it writes the block back to memory in response to P1's request. Thus the second read by P15 generates 100 stall cycles because its miss is satisfied by memory. Thus this sequence generates a total of 180 stall cycles.

For the following sequences of operations, how many stall cycles are generated by each implementation?

a. [20] <4.3>
```
P0: read 120
P0: read 128
P0: read 130
```

b. [20] <4.3>
```
P0: read 100
P0: write 108 <-- 48
P0: write 130 <-- 78
```

c. [20] <4.3>
```
P1: read 120
P1: read 128
P1: read 130
```

d. [20] <4.3>
```
P1: read 100
P1: write 108 <-- 48
P1: write 130 <-- 78
```

Parameter	Implementation 1	Implementation 2
N_{memory}	100	100
N_{cache}	70	130
$N_{invalidate}$	15	15
$N_{writeback}$	10	10

Figure 4.38 Snooping coherence latencies.

4.3 [20] <4.2> Many snooping coherence protocols have additional states, state transitions, or bus transactions to reduce the overhead of maintaining cache coherency. In Implementation 1 of Exercise 4.2, misses are incurring fewer stall cycles when they are supplied by cache than when they are supplied by memory. Some coherence protocols try to improve performance by increasing the frequency of this case.

A common protocol optimization is to introduce an Owned state (usually denoted O). The Owned state behaves like the Shared state, in that nodes may only read Owned blocks. But it behaves like the Modified state, in that nodes must supply data on other nodes' read and write misses to Owned blocks. A read miss to a block in either the Modified or Owned states supplies data to the requesting node and transitions to the Owned state. A write miss to a block in either state Modified or Owned supplies data to the requesting node and transitions to state Invalid. This optimized MOSI protocol only updates memory when a node replaces a block in state Modified or Owned.

Draw new protocol diagrams with the additional state and transitions.

4.4 [20/20/20/20] <4.2> For the following code sequences and the timing parameters for the two implementations in Figure 4.38, compute the total stall cycles for the base MSI protocol and the optimized MOSI protocol in Exercise 4.3. Assume state transitions that do not require bus transactions incur no additional stall cycles.

a. [20] <4.2>
```
P1: read 110
P15: read 110
P0: read 110
```

b. [20] <4.2>
```
P1: read 120
P15: read 120
P0: read 120
```

c. [20] <4.2>
```
P0: write 120 <-- 80
P15: read 120
P0: read 120
```

d. [20] <4.2>
```
P0: write 108 <-- 88
P15: read 108
P0: write 108 <-- 98
```

4.5 [20] <4.2> Some applications read a large data set first, then modify most or all of it. The base MSI coherence protocol will first fetch all of the cache blocks in the Shared state, and then be forced to perform an invalidate operation to upgrade them to the Modified state. The additional delay has a significant impact on some workloads.

An additional protocol optimization eliminates the need to upgrade blocks that are read and later written by a single processor. This optimization adds the Exclusive (E) state to the protocol, indicating that no other node has a copy of the block, but it has not yet been modified. A cache block enters the Exclusive state when a read miss is satisfied by memory and no other node has a valid copy. CPU reads and writes to that block proceed with no further bus traffic, but CPU writes cause the coherence state to transition to Modified. Exclusive differs from Modified because the node may silently replace Exclusive blocks (while Modified blocks must be written back to memory). Also, a read miss to an Exclusive block results in a transition to Shared, but does not require the node to respond with data (since memory has an up-to-date copy).

Draw new protocol diagrams for a MESI protocol that adds the Exclusive state and transitions to the base MSI protocol's Modified, Shared, and Invalidate states.

4.6 [20/20/20/20/20] <4.2> Assume the cache contents of Figure 4.37 and the timing of Implementation 1 in Figure 4.38. What are the total stall cycles for the following code sequences with both the base protocol and the new MESI protocol in Exercise 4.5? Assume state transitions that do not require bus transactions incur no additional stall cycles.

a. [20] <4.2>
```
P0: read 100
P0: write 100 <-- 40
```

b. [20] <4.2>
```
P0: read 120
P0: write 120 <-- 60
```

c. [20] <4.2>
```
P0: read 100
P0: read 120
```

d. [20] <4.2>
```
P0: read 100
P1: write 100 <-- 60
```

e. [20] <4.2>
```
P0: read 100
P0: write 100 <-- 60
P1: write 100 <-- 40
```

4.7 [20/20/20/20] <4.5> The test-and-set spin lock is the simplest synchronization mechanism possible on most commercial shared-memory machines. This spin lock relies on the exchange primitive to atomically load the old value and store a new value. The lock routine performs the exchange operation repeatedly until it finds the lock unlocked (i.e., the returned value is 0).

```
tas:    DADDUI   R2,R0,#1
lockit: EXCH     R2,0(R1)
        BNEZ     R2, lockit
```

Unlocking a spin lock simply requires a store of the value 0.

```
unlock: SW       R0,0(R1)
```

As discussed in Section 4.7, the more optimized test-and-test-and-set lock uses a load to check the lock, allowing it to spin with a shared variable in the cache.

```
tatas:  LD      R2, 0(R1)
        BNEZ    R2, tatas
        DADDUI  R2,R0,#1
        EXCH    R2,0(R1)
        BNEZ    R2, tatas
```

Assume that processors P0, P1, and P15 are all trying to acquire a lock at address 0x100 (i.e., register R1 holds the value 0x100). Assume the cache contents from Figure 4.37 and the timing parameters from Implementation 1 in Figure 4.38. For simplicity, assume the critical sections are 1000 cycles long.

a. [20] <4.5> Using the test-and-set spin lock, determine *approximately* how many memory stall cycles each processor incurs before acquiring the lock.

b. [20] <4.5> Using the test-and-test-and-set spin lock, determine *approximately* how many memory stall cycles each processor incurs before acquiring the lock.

c. [20] <4.5> Using the test-and-set spin lock, *approximately* how many bus transactions occur?

d. [20] <4.5> Using the test-and-test-and-set spin lock, *approximately* how many bus transactions occur?

Case Study 2: A Snooping Protocol for a Switched Network

Concepts illustrated by this case study

- Snooping Coherence Protocol Implementation
- Coherence Protocol Performance
- Coherence Protocol Optimizations
- Memory Consistency Models

The snooping coherence protocols in Case Study 1 describe coherence at an abstract level, but hide many essential details and implicitly assume atomic access to the shared bus to provide correct operation. High-performance snooping systems use one or more pipelined, switched interconnects that greatly improve bandwidth but introduce significant complexity due to transient states and nonatomic transactions. This case study examines a high-performance snooping system, loosely modeled on the Sun E6800, where multiple processor and memory nodes are connected by separate switched address and data networks.

Figure 4.39 illustrates the system organization (middle) with enlargements of a single processor node (left) and a memory module (right). Like most high-

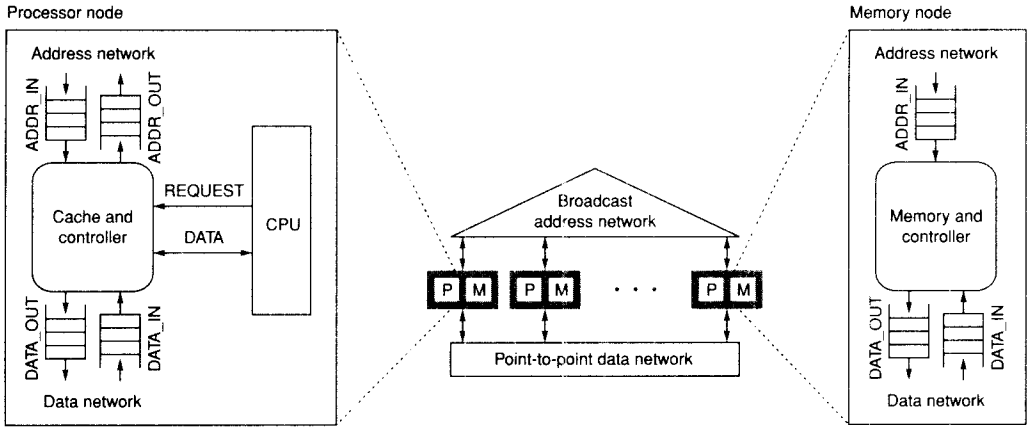

Figure 4.39 Snooping system with switched interconnect.

performance shared-memory systems, this system provides multiple memory modules to increase memory bandwidth. The processor nodes contain a CPU, cache, and a cache controller that implements the coherence protocol. The CPU issues read and write requests to the cache controller over the REQUEST bus and sends/receives data over the DATA bus. The cache controller services these requests locally, (i.e., on cache hits) and on a miss issues a coherence request (e.g., GetShared to request a read-only copy, GetModified to get an exclusive copy) by sending it to the address network via the ADDR_OUT queue. The address network uses a broadcast tree to make sure that all nodes see all coherence requests in a total order. All nodes, including the requesting node, receive this request in the same order (but not necessarily the same cycle) on the ADDR_IN queue. This total order is essential to ensure that all cache controllers act in concert to maintain coherency.

The protocol ensures that at most one node responds, sending a data message on the separate, unordered point-to-point data network.

Figure 4.40 presents a (simplified) coherence protocol for this system in tabular form. Tables are commonly used to specify coherence protocols since the multitude of states makes state diagrams too ungainly. Each row corresponds to a block's coherence state, each column represents an event (e.g., a message arrival or processor operation) affecting that block, and each table entry indicates the action and new next state (if any). Note that there are two types of coherence states. The stable states are the familiar Modified (M), Shared (S), or Invalid (I) and are stored in the cache. Transient states arise because of nonatomic transitions between stable coherence states. An important source of this nonatomicity arises because of races within the pipelined address network and between the address and data networks. For example, two cache controllers may send request messages in the same cycle for the same block, but may not find out for several cycles how the tie is broken (this is done by monitoring the ADDR_IN queue, to

State	Read	Write	Replace-ment	OwnReq	Other GetS	Other GetM	Other Inv	Other PutM	Data
I	send GetS/ IS^{AD}	send GetM/ IM^{AD}	error	error	—	—	—	—	error
S	do Read	send Inv/ SM^{A}	I	error	—	I	I	—	error
M	do Read	do Write	send PutM/ MI^{A}	error	send Data/S	send data/I	—	—	error
IS^{AD}	z	z	z	IS^{D}	—	—	—	—	save Data /IS^{A}
IM^{AD}	z	z	z	IM^{D}	—	—	—	—	save Data/IM^{A}
IS^{A}	z	z	z	do Read/S	—	—	—	—	error
IM^{A}	z	z	z	do Write/M	—	—	—	—	error
SM^{A}	z	z	z	M		II^{A}	II^{A}	—	error
MI^{A}	z	z	z	send Data/I	send Data/II^{A}	send Data/II^{A}	—	—	error
II^{A}	z	z	z	I	—	—	—	—	error
IS^{D}	z	z	z	error	—	z	z	—	save Data, do Read/S
IM^{D}	z	z	z	error	z	—	—	—	save Data, do Write/M

Figure 4.40 Broadcast snooping cache controller transitions.

see in which order the requests arrive). Cache controllers use transient states to remember what has transpired in the past while they wait for other actions to occur in the future. Transient states are typically stored in an auxiliary structure such as an MSHR, rather than the cache itself. In this protocol, transient state names encode their initial state, their intended state, and a superscript indicating which messages are still outstanding. For example, the state IS^{A} indicates that the block was in state I, wants to become state S, but needs to see its own request message (i.e., GetShared) arrive on the ADDR_IN queue before making the transition.

Events at the cache controller depend on CPU requests and incoming request and data messages. The OwnReq event means that a CPU's own request has arrived on the ADDR_IN queue. The Replacement event is a pseudo-CPU event generated when a CPU read or write triggers a cache replacement. Cache controller behavior is detailed in Figure 4.40, where each entry contains an *<action/next state>* tuple. When the current state of a block corresponds to the row of the entry and the next event corresponds to the column of the entry, then the specified action is performed and the state of the block is changed to the specified new state. If only a next state is listed, then no action is required. If no new state is listed, the state remains unchanged. Impossible cases are marked "error" and

represent error conditions. "z" means the requested event cannot currently be processed, and "—" means no action or state change is required.

The following example illustrates the basic operation of this protocol. Assume that P0 attempts a read to a block that is in state I (Invalid) in all caches. The cache controller's action—determined by the table entry that corresponds to state I and event "read"—is "send GetS/IS^{AD}," which means that the cache controller should issue a GetS (i.e., GetShared) request to the address network and transition to transient state IS^{AD} to wait for the address and data messages. In the absence of contention, P0's cache controller will normally receive its own GetS message first, indicated by the OwnReq column, causing a transition to state IS^{D}. Other cache controllers will handle this request as "Other GetS" in state I. When the memory controller sees the request on its ADDR_IN queue, it reads the block from memory and sends a data message to P0. When the data message arrives at P0's DATA_IN queue, indicated by the Data column, the cache controller saves the block in the cache, performs the read, and sets the state to S (i.e., Shared).

A somewhat more complex case arises if node P1 holds the block in state M. In this case, P1's action for "Other GetS" causes it to send the data *both* to P0 and to memory, and then transition to state S. P0 behaves exactly as before, but the memory must maintain enough logic or state to (1) not respond to P0's request (because P1 will respond) and (2) wait to respond to any future requests for this block until it receives the data from P1. This requires the memory controller to implement its own transient states (not shown). Exercise 4.11 explores alternative ways to implement this functionality.

More complex transitions occur when other requests intervene or cause address and data messages to arrive out of order. For example, suppose the cache controller in node P0 initiates a writeback of a block in state Modified. As Figure 4.40 shows, the controller does this by issuing a PutModified coherence request to the ADDR_OUT queue. Because of the pipelined nature of the address network, node P0 cannot send the data until it sees its own request on the ADDR_IN queue and determines its place in the total order. This creates an interval, called a *window of vulnerability,* where another node's request may change the action that should be taken by a cache controller. For example, suppose that node P1 has issued a GetModified request (i.e., requesting an exclusive copy) for the same block that arrives during P0's window of vulnerability for the PutModified request. In this case, P1's GetModified request logically occurs before P0's PutModified request, making it incorrect for P0 to complete the writeback. P0's cache controller must respond to P1's GetModified request by sending the block to P1 and invalidating its copy. However, P0's PutModified request remains pending in the address network, and both P0 and P1 must ignore the request when it eventually arrives (node P0 ignores the request since its copy has already been invalidated; node P1 ignores the request since the PutModified was sent by a different node).

4.8 [10/10/10/10/10/10/10] <4.2> Consider the switched network snooping protocol described above and the cache contents from Figure 4.37. What are the sequence of transient states that the affected cache blocks move through in each of the fol-

lowing cases for each of the affected caches? Assume that the address network latency is much less than the data network latency.

a. [10] <4.2> P0: read 120

b. [10] <4.2> P0: write 120 <-- 80

c. [10] <4.2> P15: write 120 <-- 80

d. [10] <4.2> P1: read 110

e. [10] <4.2> P0: write 108 <-- 48

f. [10] <4.2> P0: write 130 <-- 78

g. [10] <4.2> P15: write 130 <-- 78

4.9 [15/15/15/15/15/15/15] <4.2> Consider the switched network snooping protocol described above and the cache contents from Figure 4.37. What are the sequence of transient states that the affected cache blocks move through in each of the following cases? In all cases, assume that the processors issue their requests in the same cycle, but the address network orders the requests in top-down order. Also assume that the data network is much slower than the address network, so that the first data response arrives after all address messages have been seen by all nodes.

a. [15] <4.2> P0: read 120
P1: read 120

b. [15] <4.2> P0: read 120
P1: write 120 <-- 80

c. [15] <4.2> P0: write 120 <-- 80
P1: read 120

d. [15] <4.2> P0: write 120 <-- 80
P1: write 120 <-- 90

e. [15] <4.2> P0: replace 110
P1: read 110

f. [15] <4.2> P1: write 110 <-- 80
P0: replace 110

g. [15] <4.2> P1: read 110
P0: replace 110

4.10 [20/20/20/20/20/20/20] <4.2, 4.3> The switched interconnect increases the performance of a snooping cache-coherent multiprocessor by allowing multiple requests to be overlapped. Because the controllers and the networks are pipelined, there is a difference between an operation's latency (i.e., cycles to complete the operation) and overhead (i.e., cycles until the next operation can begin).

For the multiprocessor illustrated in Figure 4.39, assume the following latencies and overheads:

- CPU read and write hits generate no stall cycles.
- A CPU read or write that generates a replacement event issues the corresponding GetShared or GetModified message before the PutModified message (e.g., using a writeback buffer).
- A cache controller event that sends a request message (e.g., GetShared) has latency L_{send_req} and blocks the controller from processing other events for O_{send_req} cycles.
- A cache controller event that reads the cache and sends a data message has latency L_{send_data} and overhead O_{send_data} cycles.
- A cache controller event that receives a data message and updates the cache has latency L_{rcv_data} and overhead O_{rcv_data}.
- A memory controller has latency L_{read_memory} and overhead O_{read_memory} cycles to read memory and send a data message.
- A memory controller has latency L_{write_memory} and overhead O_{write_memory} cycles to write a data message to memory.
- In the absence of contention, a request message has network latency L_{req_msg} and overhead O_{req_msg} cycles.
- In the absence of contention, a data message has network latency L_{data_msg} and overhead O_{data_msg} cycles.

Consider an implementation with the performance characteristics summarized in Figure 4.41.

For the following sequences of operations and the cache contents from Figure Figure 4.37 and the implementation parameters in Figure 4.41, how many stall cycles does each processor incur for each memory request? Similarly, for how many cycles are the different controllers occupied? For simplicity, assume (1) each processor can have only one memory operation outstanding at a time, (2) if two nodes make requests in the same cycle and the one listed first "wins," the

	Implementation 1	
Action	**Latency**	**Overhead**
send_req	4	1
send_data	20	4
rcv_data	15	4
read_memory	100	20
write_memory	100	20
req_msg	8	1
data_msg	30	5

Figure 4.41 Switched snooping coherence latencies and overheads.

later node must stall for the request message overhead, and (3) all requests map to the same memory controller.

a. [20] <4.2, 4.3> `P0: read 120`

b. [20] <4.2, 4.3> `P0: write 120 <-- 80`

c. [20] <4.2, 4.3> `P15: write 120 <-- 80`

d. [20] <4.2, 4.3> `P1: read 110`

e. [20] <4.2, 4.3>
```
P0: read 120
P15: read 128
```

f. [20] <4.2, 4.3>
```
P0: read 100
P1: write 110 <-- 78
```

g. [20] <4.2, 4.3>
```
P0: write 100 <-- 28
P1: write 100 <-- 48
```

4.11 [25/25] <4.2, 4.4> The switched snooping protocol of Figure 4.40 assumes that memory "knows" whether a processor node is in state Modified and thus will respond with data. Real systems implement this in one of two ways. The first way uses a shared "Owned" signal. Processors assert Owned if an "Other GetS" or "Other GetM" event finds the block in state M. A special network ORs the individual Owned signals together; if any processor asserts Owned, the memory controller ignores the request. Note that in a nonpipelined interconnect, this special network is trivial (i.e., it is an OR gate).

However, this network becomes much more complicated with high-performance pipelined interconnects. The second alternative adds a simple directory to the memory controller (e.g., 1 or 2 bits) that tracks whether the memory controller is responsible for responding with data or whether a processor node is responsible for doing so.

a. [25] <4.2, 4.4> Use a table to specify the memory controller protocol needed to implement the second alternative. For this problem, ignore the PUTM message that gets sent on a cache replacement.

b. [25] <4.2, 4.4> Explain what the memory controller must do to support the following sequence, assuming the initial cache contents of Figure 4.37:
```
P1: read 110
P15: read 110
```

4.12 [30] <4.2> Exercise 4.3 asks you to add the Owned state to the simple MSI snooping protocol. Repeat the question, but with the switched snooping protocol above.

4.13 [30] <4.2> Exercise 4.5 asks you to add the Exclusive state to the simple MSI snooping protocol. Discuss why this is much more difficult to do with the switched snooping protocol. Give an example of the kinds of issues that arise.

4.14 [20/20/20/20] <4.6> Sequential consistency (SC) requires that all reads and writes appear to have executed in some total order. This may require the processor to stall in certain cases before committing a read or write instruction. Consider the following code sequence:

```
write A
read B
```

where the `write A` results in a cache miss and the `read B` results in a cache hit. Under SC, the processor must stall `read B` until after it can order (and thus perform) `write A`. Simple implementations of SC will stall the processor until the cache receives the data and can perform the write.

Weaker consistency models relax the ordering constraints on reads and writes, reducing the cases that the processor must stall. The Total Store Order (TSO) consistency model requires that all writes appear to occur in a total order, but allows a processor's reads to pass its own writes. This allows processors to implement write buffers, which hold committed writes that have not yet been ordered with respect to other processor's writes. Reads are allowed to pass (and potentially bypass) the write buffer in TSO (which they could not do under SC).

Assume that one memory operation can be performed per cycle and that operations that hit in the cache or that can be satisfied by the write buffer introduce no stall cycles. Operations that miss incur the latencies listed in Figure 4.41. Assume the cache contents of Figure 4.37 and the base switched protocol of Exercise 4.8. How many stall cycles occur *prior* to each operation for both the SC and TSO consistency models?

a. [20] <4.6>
```
P0: write 110 <-- 80
P0: read 108
```

b. [20] <4.6>
```
P0: write 100 <-- 80
P0: read 108
```

c. [20] <4.6>
```
P0: write 110 <-- 80
P0: write 100 <-- 90
```

d. [20] <4.6>
```
P0: write 100 <-- 80
P0: write 110 <-- 90
```

4.15 [20/20] <4.6> The switched snooping protocol above supports sequential consistency in part by making sure that reads are not performed while another node has a writeable block and writes are not performed while another processor has a writeable block. A more aggressive protocol will actually perform a write operation as soon as it receives its own GetModified request, merging the newly written word(s) with the rest of the block when the data message arrives. This may appear illegal, since another node could simultaneously be writing the block. However, the global order required by sequential consistency is determined by the order of coherence requests on the address network, so the other node's write(s) will be ordered before the requester's write(s). Note that this optimization does *not* change the memory consistency model.

Assuming the parameters in Figure 4.41:

a. [20] <4.6> How significant would this optimization be for an in-order core?

b. [20] <4.6> How significant would this optimization be for an out-of-order core?

Case Study 3: Simple Directory-Based Coherence

Concepts illustrated by this case study

- Directory Coherence Protocol Transitions
- Coherence Protocol Performance
- Coherence Protocol Optimizations

Consider the distributed shared-memory system illustrated in Figure 4.42. Each processor has a single direct-mapped cache that holds four blocks each holding two words. To simplify the illustration, the cache address tag contains the full address and each word shows only two hex characters, with the least significant word on the right. The cache states are denoted M, S, and I for Modified, Shared, and Invalid. The directory states are denoted DM, DS, and DI for Directory Modified,

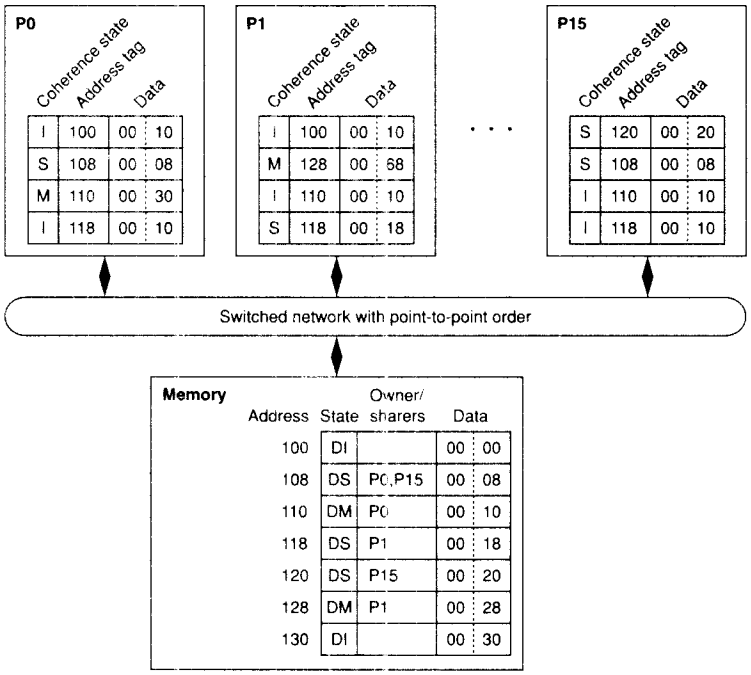

Figure 4.42 Multiprocessor with directory cache coherence.

Directory Shared, and Directory Invalid. The simple directory protocol is described in Figures 4.21 and 4.22.

4.16 [10/10/10/10/15/15/15/15] <4.4> For each part of this exercise, assume the initial cache and memory state in Figure 4.42. Each part of this exercise specifies a sequence of one or more CPU operations of the form:

```
P#: <op> <address> [ <-- <value> ]
```

where P# designates the CPU (e.g., P0), <op> is the CPU operation (e.g., read or write), <address> denotes the memory address, and <value> indicates the new word to be assigned on a write operation.

What is the final state (i.e., coherence state, tags, and data) of the caches and memory after the given sequence of CPU operations has completed? Also, what value is returned by each read operation?

a. [10] <4.4> P0: read 100

b. [10] <4.4> P0: read 128

c. [10] <4.4> P0: write 128 <-- 78

d. [10] <4.4> P0: read 120

e. [15] <4.4> P0: read 120
P1: read 120

f. [15] <4.4> P0: read 120
P1: write 120 <-- 80

g. [15] <4.4> P0: write 120 <-- 80
P1: read 120

h. [15] <4.4> P0: write 120 <-- 80
P1: write 120 <-- 90

4.17 [10/10/10/10] <4.4> Directory protocols are more scalable than snooping protocols because they send explicit request and invalidate messages to those nodes that have copies of a block, while snooping protocols broadcast all requests and invalidates to all nodes. Consider the 16-processor system illustrated in Figure 4.42 and assume that all caches not shown have invalid blocks. For each of the sequences below, identify which nodes receive each request and invalidate.

a. [10] <4.4> P0: write 100 <-- 80

b. [10] <4.4> P0: write 108 <-- 88

c. [10] <4.4> P0: write 118 <-- 90

d. [10] <4.4> P1: write 128 <-- 98

4.18 [25] <4.4> Exercise 4.3 asks you to add the Owned state to the simple MSI snooping protocol. Repeat the question, but with the simple directory protocol above.

4.19 [25] <4.4> Exercise 4.5 asks you to add the Exclusive state to the simple MSI snooping protocol. Discuss why this is much more difficult to do with the simple directory protocol. Give an example of the kinds of issues that arise.

Case Study 4: Advanced Directory Protocol

Concepts illustrated by this case study

- Directory Coherence Protocol Implementation
- Coherence Protocol Performance
- Coherence Protocol Optimizations

The directory coherence protocol in Case Study 3 describes directory coherence at an abstract level, but assumes atomic transitions much like the simple snooping system. High-performance directory systems use pipelined, switched interconnects that greatly improve bandwidth but also introduce transient states and non-atomic transactions. Directory cache coherence protocols are more scalable than snooping cache coherence protocols for two reasons. First, snooping cache coherence protocols broadcast requests to all nodes, limiting their scalability. Directory protocols use a level of indirection—a message to the directory—to ensure that requests are only sent to the nodes that have copies of a block. Second, the address network of a snooping system must deliver requests in a total order, while directory protocols can relax this constraint. Some directory protocols assume no network ordering, which is beneficial since it allows adaptive routing techniques to improve network bandwidth. Other protocols rely on point-to-point order (i.e., messages from node P0 to node P1 will arrive in order). Even with this ordering constraint, directory protocols usually have more transient states than snooping protocols. Figure 4.43 presents the cache controller state transitions for a simplified directory protocol that relies on point-to-point network ordering. Figure 4.44 presents the directory controller's state transitions. For each block, the directory maintains a state and a current owner field or a current sharers list (if any).

Like the high-performance snooping protocol presented earlier, indexing the row by the current state and the column by the event determines the *<action/next state>* tuple. If only a next state is listed, then no action is required. Impossible cases are marked "error" and represent error conditions. "z" means the requested event cannot currently be processed.

The following example illustrates the basic operation of this protocol. Suppose a processor attempts a write to a block in state I (Invalid). The corresponding tuple is "send GetM/IM^{AD}" indicating that the cache controller should send a GetM (GetModified) request to the directory and transition to state IM^{AD}. In the simplest case, the request message finds the directory in state DI (Directory Invalid), indicating that no other cache has a copy. The directory responds with a Data message that also contains the number of acks to expect (in this case zero).

State	Read	Write	Replace-ment	INV	Forwarded_ GetS	Forwarded_ GetM	PutM _Ack	Data	Last ACK
I	send GetS/ IS^D	send GetM/ IM^{AD}	error	send Ack/I	error	error	error	error	error
S	do Read	send GetM/ IM^{AD}	I	send Ack/I	error	error	error	error	error
M	do Read	do Write	send PutM/ MI^A	error	send Data, send PutMS /MS^A	send Data/I	error	error	error
IS^D	z	z	z	send Ack/ ISI^D	error	error	error	save Data, do Read/S	error
ISI^D	z	z	z	send Ack	error	error	error	save Data, do Read/I	error
IM^{AD}	z	z	z	send Ack	error	error	error	save Data /IM^A	error
IM^A	z	z	z	error	IMS^A	IMI^A	error	error	do Write/M
IMI^A	z	z	z	error	error	error	error	error	do Write, send Data/I
IMS^A	z	z	z	send Ack/ IMI^A	z	z	error	error	do Write, send Data/S
MS^A	do Read	z	z	error	send Data	send Data MI^A	/S	error	error
MI^A	z	z	z	error	send Data	send Data/I	/I	error	error

Figure 4.43 Broadcast snooping cache controller transitions.

State	GetS	GetM	PutM (owner)	PutMS (nonowner)	PutM (owner)	PutMS (nonowner)
DI	send Data, add to sharers/DS	send Data, clear sharers, set owner/ DM	error	send PutM_Ack	error	send PutM_Ack
DS	send Data, add to sharers/DS	send INVs to sharers, clear sharers, set owner, send Data/DM	error	send PutM_Ack	error	send PutM_Ack
DM	forward GetS, add to sharers/DMS^D	forward GetM, send INVs to sharers, clear sharers, set owner	save Data, send PutM_Ack/DI	send PutM_Ack	save Data, add to sharers, send PutM_Ack/ DS	send PutM_Ack
DMS^D	forward GetS, add to sharers	forward GetM, send INVs to sharers, clear sharers, set owner/ DM	save Data, send PutM_Ack/DS	send PutM_Ack	save Data, add to sharers, send PutM_Ack/ DS	send PutM_Ack

Figure 4.44 Directory controller transitions.

In this simplified protocol, the cache controller treats this single message as two messages: a Data message, followed by a Last Ack event. The Data message is processed first, saving the data and transitioning to IM^A. The Last Ack event is then processed, transitioning to state M. Finally, the write can be performed in state M.

If the GetM finds the directory in state DS (Directory Shared), the directory will send Invalidate (INV) messages to all nodes on the sharers list, send Data to the requester with the number of sharers, and transition to state M. When the INV messages arrive at the sharers, they will either find the block in state S or state I (if they have silently invalidated the block). In either case, the sharer will send an ACK directly to the requesting node. The requester will count the Acks it has received and compare that to the number sent back with the Data message. When all the Acks have arrived, the Last Ack event occurs, triggering the cache to transition to state M and allowing the write to proceed. Note that it is possible for all the Acks to arrive before the Data message, but not for the Last Ack event to occur. This is because the Data message contains the ack count. Thus the protocol assumes that the Data message is processed before the Last Ack event.

4.20 [10/10/10/10/10/10] <4.4> Consider the advanced directory protocol described above and the cache contents from Figure 4.20. What are the sequence of transient states that the affected cache blocks move through in each of the following cases?

a. [10] <4.4> `P0: read 100`

b. [10] <4.4> `P0: read 120`

c. [10] <4.4> `P0: write 120 <-- 80`

d. [10] <4.4> `P15: write 120 <-- 80`

e. [10] <4.4> `P1: read 110`

f. [10] <4.4> `P0: write 108 <-- 48`

4.21 [15/15/15/15/15/15/15] <4.4> Consider the advanced directory protocol described above and the cache contents from Figure 4.42. What are the sequence of transient states that the affected cache blocks move through in each of the following cases? In all cases, assume that the processors issue their requests in the same cycle, but the directory orders the requests in top-down order. Assume that the controllers' actions appear to be atomic (e.g., the directory controller will perform all the actions required for the DS --> DM transition before handling another request for the same block).

a. [15] <4.4> `P0: read 120`
`P1: read 120`

b. [15] <4.4> `P0: read 120`
`P1: write 120 <-- 80`

c. [15] <4.4> `P0: write 120`
`P1: read 120`

d. [15] <4.4> P0: write 120 <-- 80
P1: write 120 <-- 90

e. [15] <4.4> P0: replace 110
P1: read 110

f. [15] <4.4> P1: write 110 <-- 80
P0: replace 110

g. [15] <4.4> P1: read 110
P0: replace 110

4.22 [20/20/20/20/20] <4.4> For the multiprocessor illustrated in Figure 4.42 implementing the protocol described in Figure 4.43 and Figure 4.44, assume the following latencies:

- CPU read and write hits generate no stall cycles.
- Completing a miss (i.e., do Read and do Write) takes L_{ack} cycles *only* if it is performed in response to the Last Ack event (otherwise it gets done while the data is copied to cache).
- A CPU read or write that generates a replacement event issues the corresponding GetShared or GetModified message before the PutModified message (e.g., using a writeback buffer).
- A cache controller event that sends a request or acknowledgment message (e.g., GetShared) has latency L_{send_msg} cycles.
- A cache controller event that reads the cache and sends a data message has latency L_{send_data} cycles.
- A cache controller event that receives a data message and updates the cache has latency L_{rcv_data}.
- A memory controller incurs L_{send_msg} latency when it forwards a request message.
- A memory controller incurs an additional L_{inv} cycles for each invalidate that it must send.
- A cache controller incurs latency L_{send_msg} for each invalidate that it receives (latency is until it sends the Ack message).
- A memory controller has latency L_{read_memory} cycles to read memory and send a data message.
- A memory controller has latency L_{write_memory} to write a data message to memory (latency is until it sends the Ack message).
- A nondata message (e.g., request, invalidate, Ack) has network latency L_{req_msg} cycles
- A data message has network latency L_{data_msg} cycles.

Consider an implementation with the performance characteristics summarized in Figure 4.45.

	Implementation 1
Action	**Latency**
send_msg	6
send_data	20
rcv_data	15
read_memory	100
write_memory	20
inv	1
ack	4
req_msg	15
data_msg	30

Figure 4.45 Directory coherence latencies.

For the sequences of operations below, the cache contents of Figure 4.42, and the directory protocol above, what is the latency observed by each processor node?

a. [20] <4.4> `P0: read 100`

b. [20] <4.4> `P0: read 128`

c. [20] <4.4> `P0: write 128 <-- 68`

d. [20] <4.4> `P0: write 120 <-- 50`

e. [20] <4.4> `P0: write 108 <-- 80`

4.23 [20] <4.4> In the case of a cache miss, both the switched snooping protocol described earlier and the directory protocol in this case study perform the read or write operation as soon as possible. In particular, they do the operation as part of the transition to the stable state, rather than transitioning to the stable state and simply retrying the operation. This is *not* an optimization. Rather, to ensure forward progress, protocol implementations must ensure that they perform at least one CPU operation before relinquishing a block.

Suppose the coherence protocol implementation didn't do this. Explain how this might lead to livelock. Give a simple code example that could stimulate this behavior.

4.24 [20/30] <4.4> Some directory protocols add an Owned (O) state to the protocol, similar to the optimization discussed for snooping protocols. The Owned state behaves like the Shared state, in that nodes may only read Owned blocks. But it behaves like the Modified state, in that nodes must supply data on other nodes' Get requests to Owned blocks. The Owned state eliminates the case where a GetShared request to a block in state Modified requires the node to send the data both to the requesting processor and to the memory. In a MOSI directory protocol, a GetShared request to a block in either the Modified or Owned states supplies data to the requesting node and transitions to the Owned state. A GetModified request in

state Owned is handled like a request in state Modified. This optimized MOSI protocol only updates memory when a node replaces a block in state Modified or Owned.

a. [20] <4.4> Explain why the MS^A state in the protocol is essentially a "transient" Owned state.

b. [30] <4.4> Modify the cache and directory protocol tables to support a stable Owned state.

4.25 [25/25] <4.4> The advanced directory protocol described above relies on a point-to-point ordered interconnect to ensure correct operation. Assuming the initial cache contents of Figure 4.42 and the following sequences of operations, explain what problem could arise if the interconnect failed to maintain point-to-point ordering. Assume that the processors perform the requests at the same time, but they are processed by the directory in the order shown.

a. [25] <4.4>
```
P1: read 110
P15: write 110 <-- 90
```

b. [25] <4.4>
```
P1: read 110
P0: replace 110
```

5.1	Introduction	288
5.2	Eleven Advanced Optimizations of Cache Performance	293
5.3	Memory Technology and Optimizations	310
5.4	Protection: Virtual Memory and Virtual Machines	315
5.5	Crosscutting Issues: The Design of Memory Hierarchies	324
5.6	Putting It All Together: AMD Opteron Memory Hierarchy	326
5.7	Fallacies and Pitfalls	335
5.8	Concluding Remarks	341
5.9	Historical Perspective and References	342
	Case Studies with Exercises by Norman P. Jouppi	342