

## 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 Developer Forum in 2005*

## 5.1

**Introduction**

As the quotations that open this chapter show, 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 1986–2003, 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 instruction-level parallelism (ILP) combined with growing concern over power, is leading to a new era in computer architecture—an era where multiprocessors play a major role from the low end to the high end. The second quotation captures this clear inflection point.

This increased importance of multiprocessing reflects several major factors:

- The dramatically lower efficiencies in silicon and energy use that were encountered between 2000 and 2005 as designers attempted to find and exploit more ILP, which turned out to be inefficient, since power and silicon costs grew faster than performance. Other than ILP, the only scalable and general-purpose way we know how to increase performance faster than the basic technology allows (from a switching perspective) is through multiprocessing.
- A growing interest in high-end servers as cloud computing and software-as-a-service become more important.
- A growth in data-intensive applications driven by the availability of massive amounts of data on the Internet.
- The insight that increasing performance on the desktop is less important (outside of graphics, at least), either because current performance is acceptable or because highly compute- and data-intensive applications are being done in the cloud.
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant natural parallelism, arising from large datasets, natural parallelism (which occurs in scientific codes), or parallelism among large numbers of independent requests (request-level parallelism).
- The advantages of leveraging a design investment by replication rather than unique design; all multiprocessor designs provide such leverage.

In this chapter, we focus on exploiting thread-level parallelism (TLP). TLP implies the existence of multiple program counters and hence is exploited primarily

through MIMDs. Although MIMDs have been around for decades, the movement of thread-level parallelism to the forefront across the range of computing from embedded applications to high-end servers is relatively recent. Likewise, the extensive use of thread-level parallelism for general-purpose applications, versus scientific applications, is relatively new.

Our focus in this chapter is on *multiprocessors*, which we define as computers consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space. Such systems exploit thread-level parallelism through two different software models. The first is the execution of a tightly coupled set of threads collaborating on a single task, which is typically called *parallel processing*. The second is the execution of multiple, relatively independent processes that may originate from one or more users, which is a form of *request-level parallelism*, although at a much smaller scale than what we explore in the next chapter. Request-level parallelism may be exploited by a single application running on multiple processors, such as a database responding to queries, or multiple applications running independently, often called *multiprogramming*.

The multiprocessors we examine in this chapter typically range in size from a dual processor to dozens of processors and communicate and coordinate through the sharing of memory. Although sharing through memory implies a shared address space, it does not necessarily mean there is a single physical memory. Such multiprocessors include both single-chip systems with multiple cores, known as *multicore*, and computers consisting of multiple chips, each of which may be a multicore design.

In addition to true multiprocessors, we will return to the topic of multithreading, a technique that supports multiple threads executing in an interleaved fashion on a single multiple issue processor. Many multicore processors also include support for multithreading.

In the next chapter, we consider ultrascale computers built from very large numbers of processors, connected with networking technology and often called *clusters*; these large-scale systems are typically used for cloud computing with a model that assumes either massive numbers of independent requests or highly parallel, intensive compute tasks. When these clusters grow to tens of thousands of servers and beyond, we call them *warehouse-scale computers*.

In addition to the multiprocessors we study here and the warehouse-scaled systems of the next chapter, there are a range of special large-scale multiprocessor systems, sometimes called *multicomputers*, which are less tightly coupled than the multiprocessors examined in this chapter but more tightly coupled than the warehouse-scale systems of the next. The primary use for such multicomputers is in high-end scientific computation. Many other books, such as Culler, Singh, and Gupta [1999], cover such systems in detail. Because of the large and changing nature of the field of multiprocessing (the just-mentioned Culler et al. reference is over 1000 pages and discusses only multiprocessing!), we have chosen to focus our attention on what we believe is the most important and general-purpose portions of the computing space. Appendix I discusses some of the issues that arise in building such computers in the context of large-scale scientific applications.

Thus, our focus will be on multiprocessors with a small to moderate number of processors (2 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 (33 or more processors), primarily in Appendix I, which covers more aspects of the design of such processors, as well as the behavior performance for parallel scientific workloads, a primary class of applications for large-scale multiprocessors. In large-scale multiprocessors, the interconnection networks are a critical part of the design; Appendix F focuses on that topic.

## Multiprocessor Architecture: Issues and Approach

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 operating system (from multiple independent requests). 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 or programmer 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 with an SIMD processor or with a GPU (see Chapter 4). This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor or GPU 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 in an MIMD.

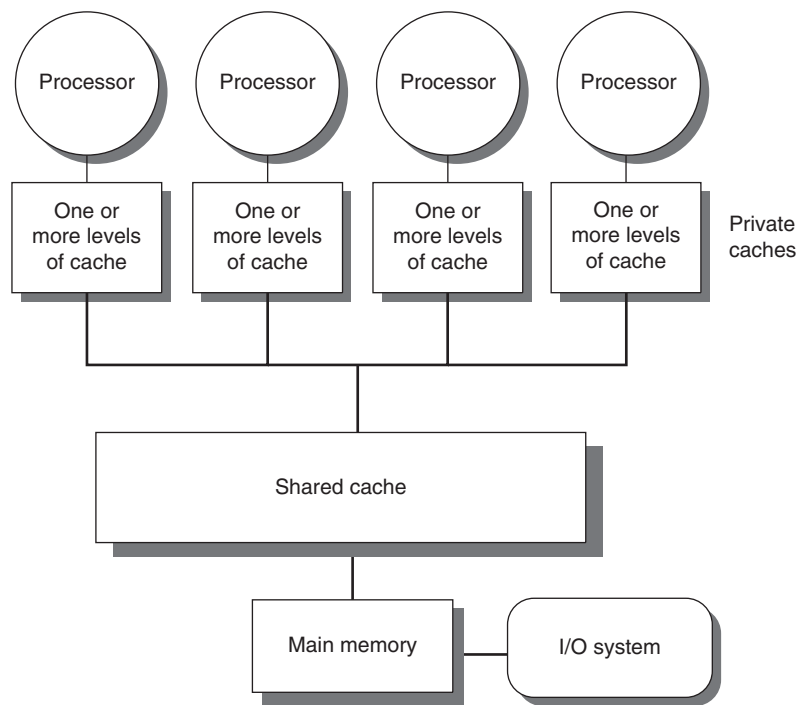
Existing shared-memory multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors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

The first group, which we call *symmetric (shared-memory) multiprocessors* (SMPs), or *centralized shared-memory multiprocessors*, features small numbers of cores, typically eight or fewer. For multiprocessors with such small processor counts, it is possible for the processors to share a single centralized memory that all processors have equal access to, hence the term *symmetric*. In multicore chips, the memory is effectively shared in a centralized fashion among the cores, and all existing multicores are SMPs. When more than one multicore is connected, there are separate memories for each multicore, so the memory is distributed rather than centralized.

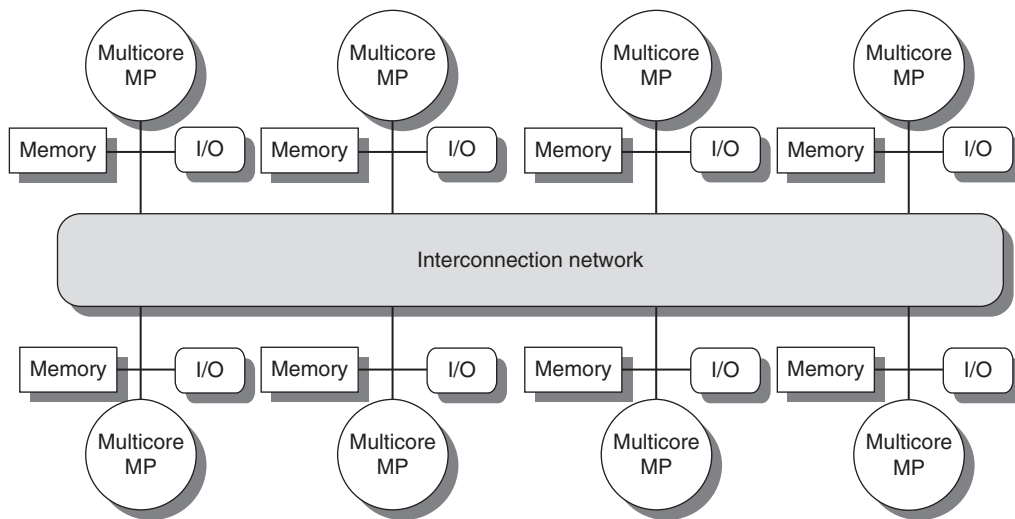
SMP architectures are also sometimes called *uniform memory access* (UMA) multiprocessors, arising from the fact that all processors have a uniform latency

from memory, even if the memory is organized into multiple banks. Figure 5.1 shows what these multiprocessors look like. The architecture of SMPs is the topic of Section 5.2, and we explain the approach in the context of a multicore.

The alternative design approach consists of multiprocessors with physically distributed memory, called *distributed shared memory* (DSM). Figure 5.2 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 introduction of multicore processors has meant that even two-chip multiprocessors use distributed memory. The larger number of processors also raises the need for a high-bandwidth interconnect, of which we will see examples in Appendix F. Both



**Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip.** Multiple processor–cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.



**Figure 5.2** The basic architecture of a distributed-memory multiprocessor in 2011 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes. Each processor core shares the entire memory, although the access time to the lock memory attached to the core's chip will be much faster than the access time to remote memories.

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

Distributing the memory among the nodes both increases the bandwidth and reduces the latency to local memory. A DSM multiprocessor is also called a *NUMA* (nonuniform memory access), since the access time depends on the location of a data word in memory. The key disadvantages for a DSM are that communicating data among processors becomes somewhat more complex, and a DSM requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories. Because all multicore-based multiprocessors with more than one processor chip (or socket) use distributed memory, we will explain the operation of distributed memory multiprocessors from this viewpoint.

In both SMP and DSM architectures, communication among threads occurs through a 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. The term *shared memory* associated with both SMP and DSM refers to the fact that the *address space* is shared.

In contrast, the clusters and warehouse-scale computers of the next chapter look like individual computers connected by a network, and the memory of one processor cannot be accessed by another processor without the assistance of software protocols running on both processors. In such designs, message-passing protocols are used to communicate data among processors.

## 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** Recall from Chapter 1 that Amdahl's law is

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

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 the 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 separate cores may cost 35 to 50 clock cycles and among cores on separate chips anywhere from 100 clock cycles to as much as 500 or more 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 3.3 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 clock cycles per instruction. 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

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{200 \text{ ns}}{0.3 \text{ ns}} = 666 \text{ cycles}$$

Hence, we can compute the CPI:

$$\text{CPI} = 0.5 + 1.2 = 1.7$$

The multiprocessor with all local references is  $1.7/0.5 = 3.4$  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 offer better parallel performance, as well as by software systems that maximize the amount of time spent executing with the full



complement of processors. 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 multi-threading (discussed later in this chapter) or by using prefetching (a topic we cover extensively in Chapter 2).

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, Sections 5.2 through 5.4 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 5.5 discusses synchronization, which, because it inherently involves interprocessor communication and also can limit parallelism, is a major potential bottleneck. Section 5.6 covers latency-hiding techniques and memory consistency models for shared memory. In Appendix I, we focus primarily on larger-scale multiprocessors that 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.

## 5.2

## Centralized Shared-Memory Architectures

The observation that the use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor is the key insight that motivates centralized memory multiprocessors. Originally, these processors were all single-core and often took an entire board, and memory was located on a shared bus. With more recent, higher-performance processors, the memory demands have outstripped the capability of reasonable buses, and recent microprocessors directly connect memory to a single chip, which is sometimes called a *backside* or *memory bus* to distinguish it from the bus used to connect to I/O. Accessing a chip's local memory whether for an I/O operation or for an access from another chip requires going through the chip that "owns" that memory. Thus, access to memory is asymmetric: faster to the local memory and slower to the remote memory. In a multicore that memory is shared among all the cores on a single chip, but the asymmetric access to the memory of one multicore from the memory of another remains.

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 5.3 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem*. Notice that the coherence problem exists because we have both a global state, defined primarily by the main memory, and a local state, defined by the individual caches, which are private to each processor core. Thus, in a multicore where some level of caching may be shared (for example, an L3), while some levels are private (for example, L1 and L2), the coherence problem still exists and must be solved.

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 processor P to 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.

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

**Figure 5.3** 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!

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 processors 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 5.6](#).

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 5.6](#), 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, 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 in use, each of which uses different techniques to track the sharing status:

- *Directory based*—The sharing status of a particular block of physical memory is kept in one location, called the *directory*. There are two very different types of directory-based cache coherence. In an SMP, we can use one *centralized directory*, associated with the memory or some other single serialization point, such as the outermost cache in a multicore. In a DSM, it makes no sense to have a single directory, since that would create a single point of contention and make it difficult to scale to many multicore chips given the memory demands of multicores with eight or more cores. Distributed directories are more complex than a single directory, and such designs are the subject of Section 5.4.
- *Snooping*—Rather than keeping the state of sharing in a single directory, every cache that has a copy of the data from a block of physical memory could track the sharing status of the block. In an SMP, the caches are typically all accessible via some broadcast medium (e.g., a bus connects the per-core caches to the shared cache or memory), 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. Snooping can also be used as the coherence protocol for a multichip multiprocessor, and some designs support a snooping protocol on top of a directory protocol within each multicore!

Snooping protocols became popular with multiprocessors using microprocessors (single-core) and caches attached to a single shared memory by a bus.

The bus provided a convenient broadcast medium to implement the snooping protocols. Multicore architectures changed the picture significantly, since all multicores share some level of cache on the chip. Thus, some designs switched to using directory protocols, since the overhead was small. To allow the reader to become familiar with both types of protocols, we focus on a snooping protocol here and discuss a directory protocol when we come to DSM architectures.

## Snooping Coherence 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. 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 5.4 shows an example of an invalidation protocol 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 to write

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

**Figure 5.4** 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 processor 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, processor 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. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 for each core. It is this observation that led some designers to opt for a directory protocol within the multicore. To make this work the L3 cache must be inclusive (see page 397).

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, 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 multicore is the use of the bus, or another broadcast medium, to perform invalidates. In older multiple-chip multiprocessors, the bus used for coherence is the shared-memory access bus. In a multicore, the bus can be the connection between the private caches (L1 and L2 in the Intel Core i7) and the shared outer cache (L3 in the i7). 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 and must effectively be treated as additional cache entries.)

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 private cache rather than in the shared cache or 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 shared 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 (or L3) access to be aborted. The additional complexity comes from having to retrieve the cache block from another processor's private cache (L1 or L2), which can often take longer than retrieving it from L3. Since write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors. As a result, all multicore processors use write-back at the outermost levels of the cache, and 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 would 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 to 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 core for that block. The core 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 and have snoop accesses directed to the duplicate tags. Another approach is to use a directory at the shared L3 cache; the directory indicates whether a given block is shared and possibly which cores have copies. With the directory information, invalidates can be directed only to those caches with copies of the cache block. This requires that L3 must always have a copy of any data item in L1 or L2, a property called *inclusion*, which we will return to in [Section 5.7](#).

## An Example Protocol

A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each core. This controller responds to requests from the processor in the core 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 private 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 in the private cache is potentially shared, while the modified state indicates that the block has been updated in the private cache; note that the modified state *implies* that the block is exclusive. Figure 5.5 shows the requests generated by a core (in the top half of the table)

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 local 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 local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or <i>ownership</i> 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 shared cache or 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 shared block; 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 in the local cache.

**Figure 5.5** The cache coherence mechanism receives requests from both the core's processor and the shared bus and responds to these based on the type of request, whether it hits or misses in the local cache, and the state of the local 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 local cache and the block is valid.



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 private cache. We describe this and other extensions on page 362.

When an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the cache block invalidate it. For a write miss in a write-back cache, if the block is exclusive in just one private cache, that cache also writes back the block; otherwise, the data can be read from the shared cache or memory.

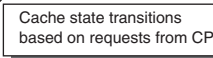
Figure 5.6 shows a finite-state transition diagram for a single private cache block using a write invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top half of the table in Figure 5.5), as opposed to transitions based on bus requests (on the right, which corresponds to the bottom half of the table in Figure 5.5). 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 private 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 5.6 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 5.6 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 5.7 shows how the state transitions in the right half of Figure 5.6 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 private 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 local caches to make the block invalid. In addition, if some other local cache had the block in exclusive state, that local 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 local cache with the exclusive copy changes its state to shared.

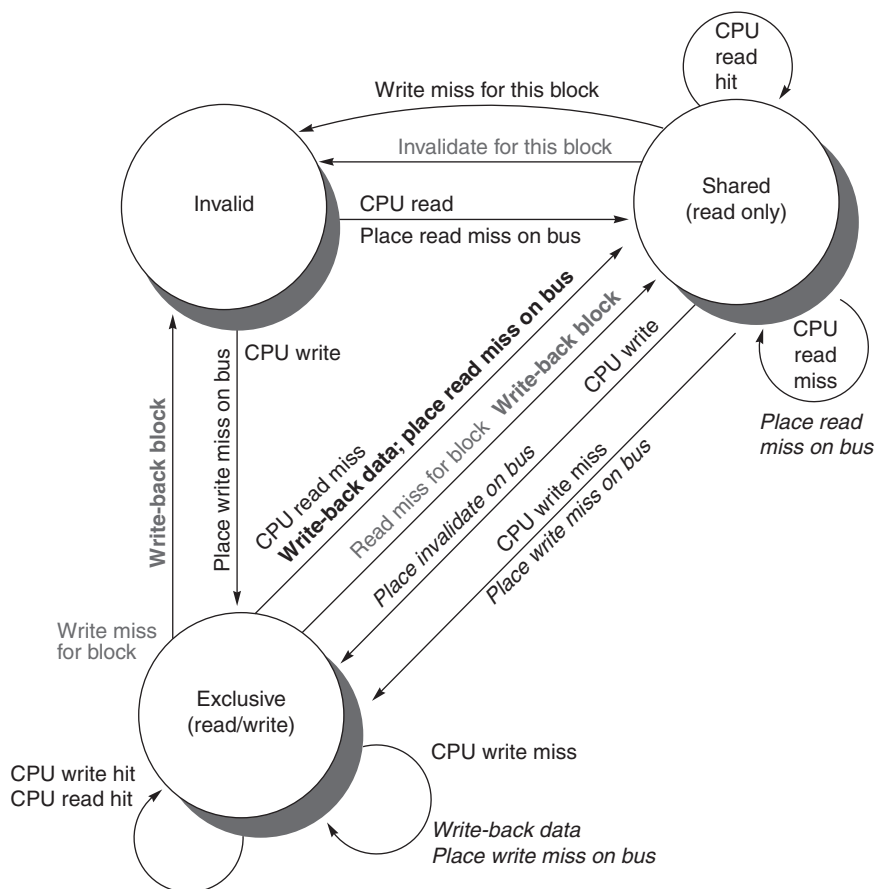
The actions in gray in Figure 5.7, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the outer shared cache (L2 or L3,



**Figure 5.6 A write invalidate, cache coherence protocol for a private 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 local 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 private 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 local 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 private 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 (or a shared cache) provides data on a read miss for a block that is clean in all local 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. In a multicore chip, the shared cache (usually L3, but sometimes L2) acts as the equivalent of memory, and the bus is the bus between the private caches of each core and the shared cache, which in turn interfaces to the memory.

or memory if there is no shared cache), which simplifies the implementation. In fact, it does not matter whether the level out from the private caches is a shared cache or memory; the key is that all accesses from the cores go through that level.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and



**Figure 5.7** 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 5.6, the activities on a transition are shown in bold.

receive a response as a single atomic action. In reality this is not true. In fact, even a read miss might not be atomic; after detecting a miss in the L2 of a multi-core, the core must arbitrate for access to the bus connecting to the shared L3. Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. We will explore these complications later in this section and when we examine DSM designs.

With multicore processors, the coherence among the processor cores is all implemented on chip, using either a snooping or simple central directory protocol. Many dual-processor chips, including the Intel Xeon and AMD Opteron, supported multichip multiprocessors that could be built by connecting a high-speed interface (called Quickpath or Hypertransport, respectively). These next-level interconnects are not just extensions of the shared bus, but use a different approach for interconnecting multicores.

A multiprocessor built with multiple multicore chips will have a distributed memory architecture and will need an interchip coherency mechanism above and beyond the one within the chip. In most cases, some form of directory scheme is used.

## Extensions to the Basic Coherence Protocol

The coherence protocol we have just described is a simple three-state protocol and is often referred to by the first letter of the states, making it a MSI (Modified, Shared, Invalid) protocol. There are many extensions of this basic protocol, which we mentioned in the captions of figures in this section. These extensions are created by adding additional states and transactions, which optimize certain behaviors, possibly resulting in improved performance. Two of the most common extensions are

1. *MESI* adds the state Exclusive to the basic MSI protocol to indicate when a cache block is resident only in a single cache but is clean. If a block is in the E state, it can be written without generating any invalidates, which optimizes the case where a block is read by a single cache before being written by that same cache. Of course, when a read miss to a block in the E state occurs, the block must be changed to the S state to maintain coherence. 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 core need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this local 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 block is modified. The popular MESI protocol, which is named for the four states it includes (Modified, Exclusive, Shared, and Invalid), uses this structure. The Intel i7 uses a variant of a MESI protocol, called MESIF, which adds a state (Forward) to designate which sharing processor should respond to a request. It is designed to enhance performance in distributed memory organizations.
2. *MOESI* adds the state Owned to the MESI protocol to indicate that the associated block is owned by that cache and out-of-date in memory. In MSI and MESI protocols, when there is an attempt to share a block in the Modified state, the state is changed to Shared (in both the original and newly sharing cache), and the block must be written back to memory. In a MOESI protocol, the block can be changed from the Modified to Owned state in the original cache without writing it to memory. Other caches, which are newly sharing the block, keep the block in the Shared state; the O state, which only the original cache holds, indicates that the main memory copy is out of date and that the designated cache is the owner. The owner of the block must supply it on a miss, since memory is not up to date and must write the block back to memory if it is replaced. The AMD Opteron uses the MOESI protocol.

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.

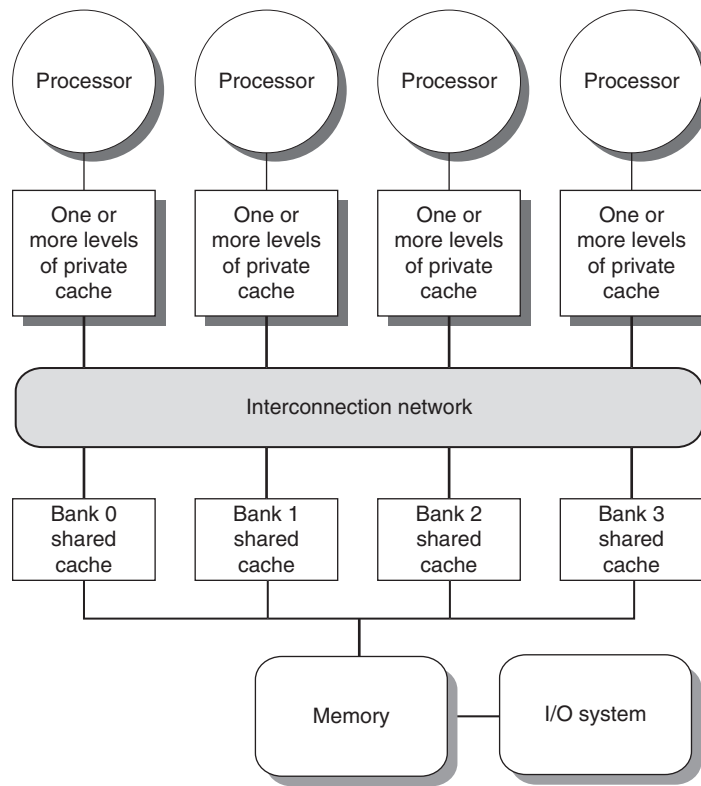
### Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

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. Using the higher bandwidth connection available on-chip and a shared L3 cache, which is faster than memory, designers have managed to support four to eight high-performance cores in a symmetric fashion. Such an approach is unlikely to scale much past eight cores, and it will not work once multiple multicores are combined.

Snooping bandwidth at the caches can also become a problem, since every cache must examine every miss placed on the bus. As we mentioned, duplicating the tags is one solution. Another approach, which has been adopted in some recent multicores, is to place a directory at the level of the outermost cache. The directory explicitly indicates which processor's caches have copies of every item in the outermost cache. This is the approach Intel uses on the i7 and Xeon 7000 series. Note that the use of this directory does not eliminate the bottleneck due to a shared bus and L3 among the processors, but it is much simpler to implement than the distributed directory schemes that we will examine in Section 5.4.

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 (either main memory or a shared cache) can be configured into multiple physical banks, so as to boost the effective memory bandwidth while retaining uniform access time to memory. Figure 5.8 shows how such a system might look if it were implemented with a single-chip multicore. Although such an approach might be used to allow more than four cores to be interconnected on a single chip, it does not scale well to a multichip multiprocessor that uses multicore building blocks, since the memory is already attached to the individual multicore chips, rather than centralized.

The AMD Opteron represents another intermediate point in the spectrum between a snooping and a directory protocol. Memory is directly connected to each multicore chip, and up to four multicore chips can be connected. The system is a NUMA, since local memory is somewhat faster. 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 completed is by an explicit acknowledgment. Thus, the coherence protocol uses a broadcast to



**Figure 5.8** A multicore single-chip multiprocessor with uniform memory access through a banked shared cache and using an interconnection network rather than a bus.

find potentially shared copies, like a snooping protocol, but uses the acknowledgments to order operations, like a directory protocol. Because local memory is only somewhat faster than remote memory in the Opteron implementation, some software treats an Opteron multiprocessor as having uniform memory access.

A snooping 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 5.4](#), 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 snooping protocol.

## ~~Implementing Snooping 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 core of an Intel i7 in 2011! 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 2011, most multicore processors that support only a single-chip multiprocessor have opted to use a shared bus structure connecting to either a shared memory or a shared cache. In contrast, *every* multicore multiprocessor system that supports 16 or more cores 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 single multicore chip, these steps can be made effectively atomic by arbitrating for the bus to the shared cache or memory 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 some multicores, 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 snooping system, ensuring that a race has only one winner is accomplished 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 snooping cache coherence without a bus. We explain the details in Appendix I.

It is possible to combine snooping and directories, and several designs use snooping within a multicore and directories among multiple chips or, *vice versa*, directories within a multicore and snooping among multiple chips.

### 5.3

## Performance of Symmetric Shared-Memory Multiprocessors

In a multicore using a snooping 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 B 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 the 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 I, 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 when running a general-purpose commercial workload. The study we examine was done with a four-processor Alpha system in 1998, but it remains the most comprehensive and insightful study of the performance of a multiprocessor for such workloads. 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 systems designed in 2011, the basic structure of the system, consisting of a four-issue processor and a three-level cache hierarchy, is very similar to the multicore Intel i7 and other processors, as shown in [Figure 5.9](#). In particular, the Alpha caches are somewhat smaller, but the miss times are also lower than on an i7. Thus, the behavior of the Alpha system should provide interesting insights into the behavior of modern multicore designs.

The workload used for this study consists of three applications:

1. An online transaction-processing (OLTP) workload modeled after TPC-B (which has memory behavior similar to its newer cousin TPC-C, described in [Chapter 1](#)) 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 processor 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, the older cousin of the heavily used TPC-E, which also uses Oracle 7.3.2 as the underlying database. The workload includes only 6 of the 17 read queries in TPC-D,

Cache level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I/8 KB D	32 KB I/32 KB D
	Associativity	Direct mapped	4-way I/8-way D
	Block size	32 B	64 B
	Miss penalty	7	10
L2	Size	96 KB	256 KB
	Associativity	3-way	8-way
	Block size	32 B	64 B
	Miss penalty	21	35
L3	Size	2 MB	2 MB per core
	Associativity	Direct mapped	16-way
	Block size	64 B	64 B
	Miss penalty	80	~100

**Figure 5.9** The characteristics of the cache hierarchy of the Alpha 21164 used in this study and the Intel i7. Although the sizes are larger and the associativity is higher on the i7, the miss penalties are also higher, so the behavior may differ only slightly. For example, from Appendix B, we can estimate the miss rates of the smaller Alpha L1 cache as 4.9% and 3% for the larger i7 L1 cache, so the average L1 miss penalty per reference is 0.34 for the Alpha and 0.30 for the i7. Both systems have a high penalty (125 cycles or more) for a transfer required from a private cache. The i7 also shares its L3 among all the cores.

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

**Figure 5.10** The distribution of execution time in the commercial workloads. The OLTP benchmark has the largest fraction of both OS time and processor 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, Gharachorloo, and Bugnion [1998] on a four-processor AlphaServer 4100.

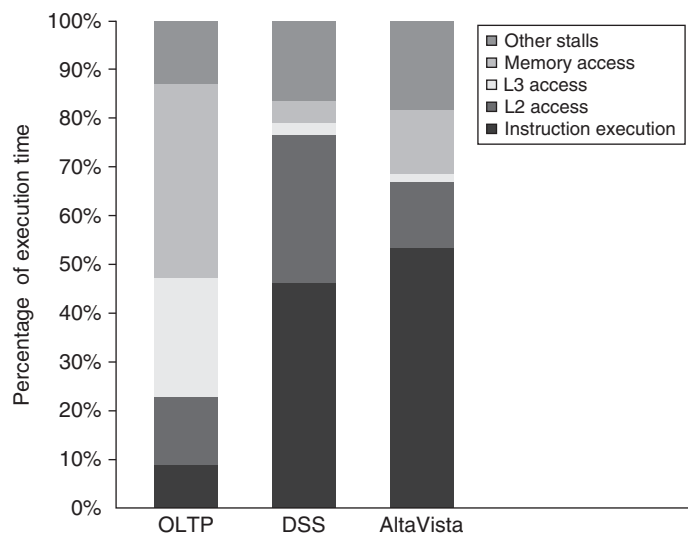
although the 6 queries examined in the benchmark span the range of activities in the entire benchmark. To hide the I/O latency, parallelism is exploited both within queries, where parallelism is detected during a query formulation process, and across queries. Blocking calls are much less frequent than in the OLTP benchmark; the 6 queries average about 1.5 million instructions before blocking.

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. AltaVista was the most popular Web search engine before the arrival of Google.

Figure 5.10 shows the percentages of time spent in user mode, in the kernel, and in the idle loop. 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 processor execution for these benchmarks on the four-processor system; as discussed on page 367, these benchmarks include substantial I/O time, which is ignored in the processor 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 5.11 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 translation lookaside buffer (TLB) and branch mispredict stalls). Although the performance of the DSS

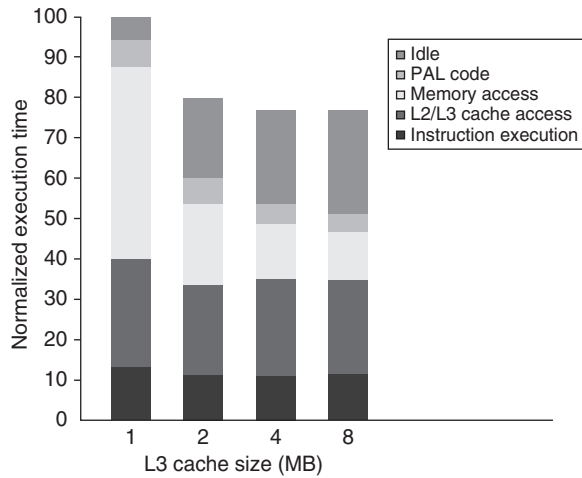


**Figure 5.11** The execution time breakdown for the three programs (OLTP, DSS, and AltaVista) in the commercial workload. The DSS numbers are the average across six different queries. The CPI varies widely from a low of 1.3 for AltaVista, to 1.61 for the DSS queries, to 7.0 for OLTP. (Individually, 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].

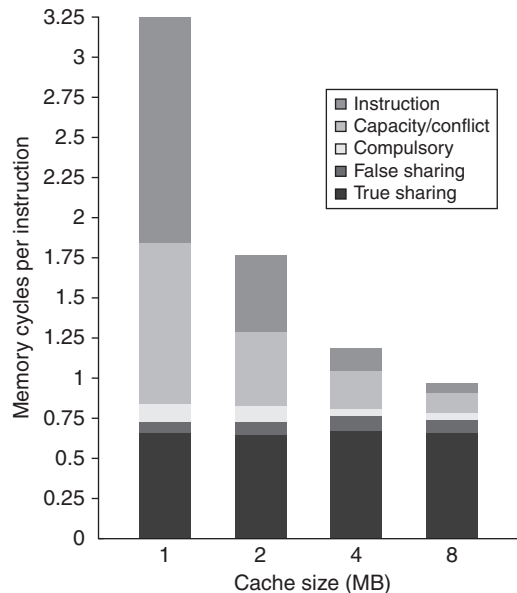
and AltaVista workloads is reasonable, the performance of the OLTP workload is very poor, due to a poor performance of the memory hierarchy.

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 5.12](#) 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, 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?

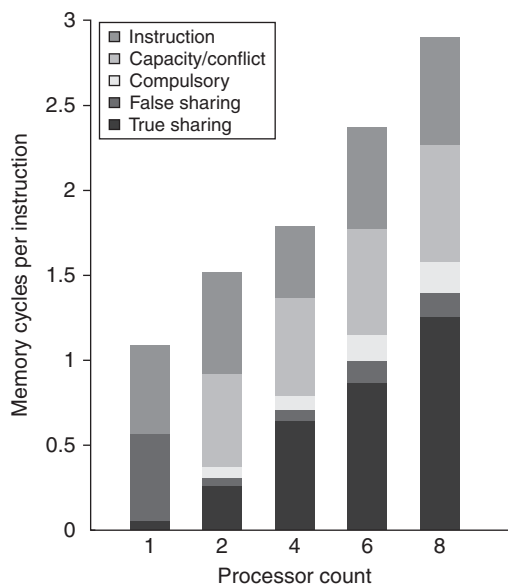
To better understand the answer to this question, we need to determine what factors contribute to the L3 miss rate and how they change as the L3 cache grows. [Figure 5.13](#) shows these data, displaying the number of memory access cycles contributed per instruction from five sources. The two largest sources of L3 memory access cycles with a 1 MB L3 are instruction and capacity/conflict



**Figure 5.12** 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. The PAL code is a set of sequences of specialized OS-level instructions executed in privileged mode; an example is the TLB miss handler.



**Figure 5.13** The contributing causes of memory access cycle shift as the cache size is increased. The L3 cache is simulated as two-way set associative.



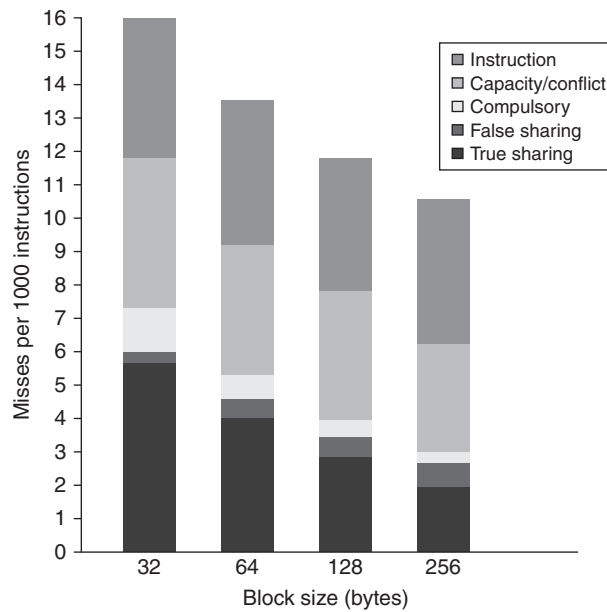
**Figure 5.14** The contribution to memory access cycles increases as processor count increases primarily due to increased true sharing. The compulsory misses slightly increase since each processor must now handle more compulsory misses.

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.

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 5.14 shows these 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 5.15 shows the number of misses per 1000 instructions as the block size is increased from 32 to 256 bytes. Increasing the block size from 32 to 256 bytes 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.
- The compulsory miss rate significantly decreases, as we would expect.



**Figure 5.15** 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.

- 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. Based on these data, the miss penalty for a larger block size L3 to perform as well as the 32-byte block size L3 can be expressed as a multiplier on the 32-byte block size penalty:

Block size	Miss penalty relative to 32-byte block miss penalty
64 bytes	1.19
128 bytes	1.36
256 bytes	1.52

With modern DDR SDRAMs that make block access fast, these numbers seem attainable, especially at the 128 byte block size. Of course, we must also worry about the effects of the increased traffic to memory and possible contention for the memory with other cores. This latter effect may easily negate the gains obtained from improving the performance of a single processor.

## 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 version of the Unix “make” command executed 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:

- *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, 10 clock cycle hit time.
- *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 5.16 shows how the execution time breaks down for the eight processors using the parameters just listed. Execution time is broken down into four components:

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



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

**Figure 5.16** 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.

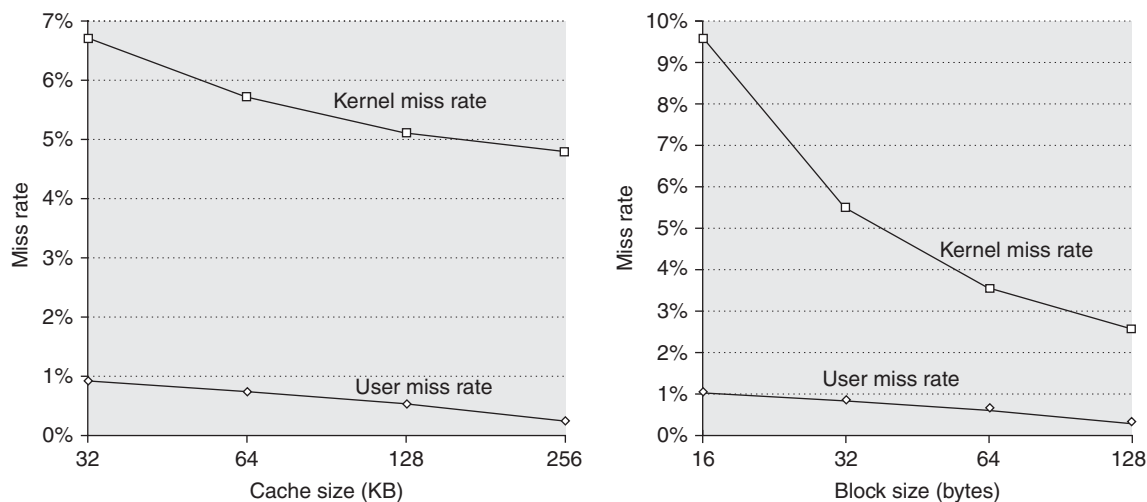
This multiprogramming workload has a significant instruction cache performance loss, at least for the OS. The instruction cache miss rate in the OS for a 64-byte block size, two-way set associative cache varies from 1.7% for a 32 KB cache to 0.2% for a 256 KB cache. User-level instruction cache misses 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 5.17 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

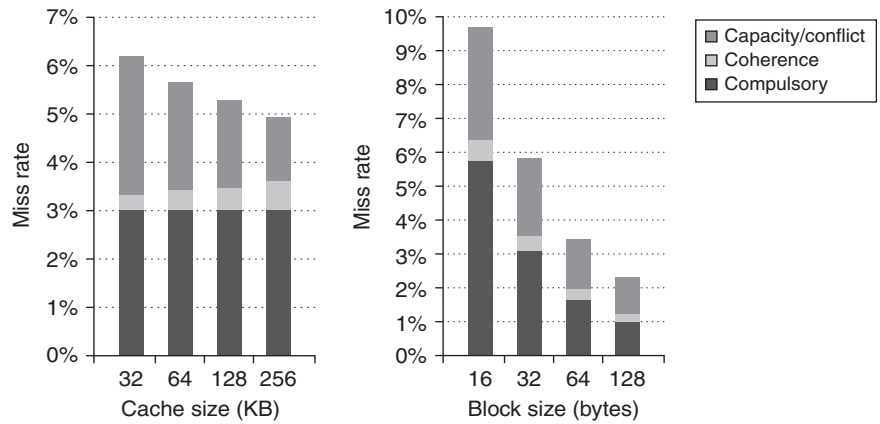


**Figure 5.17** The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 KB to 256 KB (with a 32-byte block) causes the user miss 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).

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 how the kernel misses behave.

Figure 5.18 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 5.18 confirms that, for the kernel references, increasing the cache size reduces only 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.

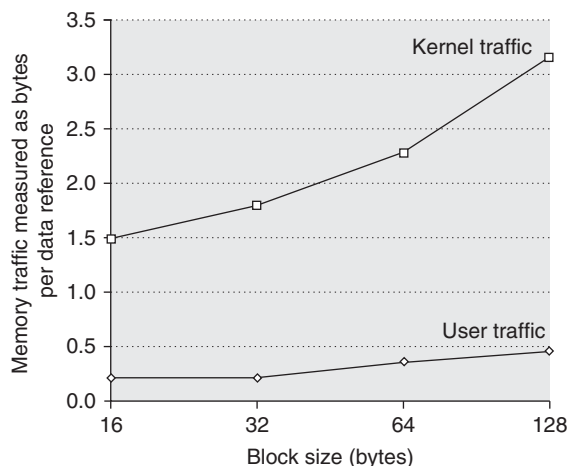
If we examine the number of bytes needed per data reference, as in Figure 5.19, we see that the kernel has a higher traffic ratio that grows with block size. It is easy to see why this occurs: When going from a 16-byte block to a 128-byte block, the miss rate drops by about 3.7, but the number of bytes



**Figure 5.18** 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 no 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%.

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, and the newest



**Figure 5.19** 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 I.

AMD processors have similar support. It is harder to detect such behavior by the OS, and doing so may require programmer assistance, but the payoff is potentially even greater.

OS and commercial workloads pose tough challenges for multiprocessor memory systems, and unlike scientific applications, which we examine in Appendix I, they are less amenable to algorithmic or compiler restructuring. As the number of cores increases predicting the behavior of such applications is likely to get more difficult. Emulation or simulation methodologies that allow the simulation of hundreds of cores with large applications (including operating systems) will be crucial to maintaining an analytical and quantitative approach to design.

## 5.4

### Distributed Shared-Memory and Directory-Based Coherence

As we saw in [Section 5.2](#), 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, consider a multiprocessor composed of four 4-core multicores capable of sustaining one data reference per clock and a 4 GHz clock. From the data in Section I.5 of Appendix I, we can see that the applications may require 4 GB/sec to 170 GB/sec of bus bandwidth. Although the caches in those experiments are

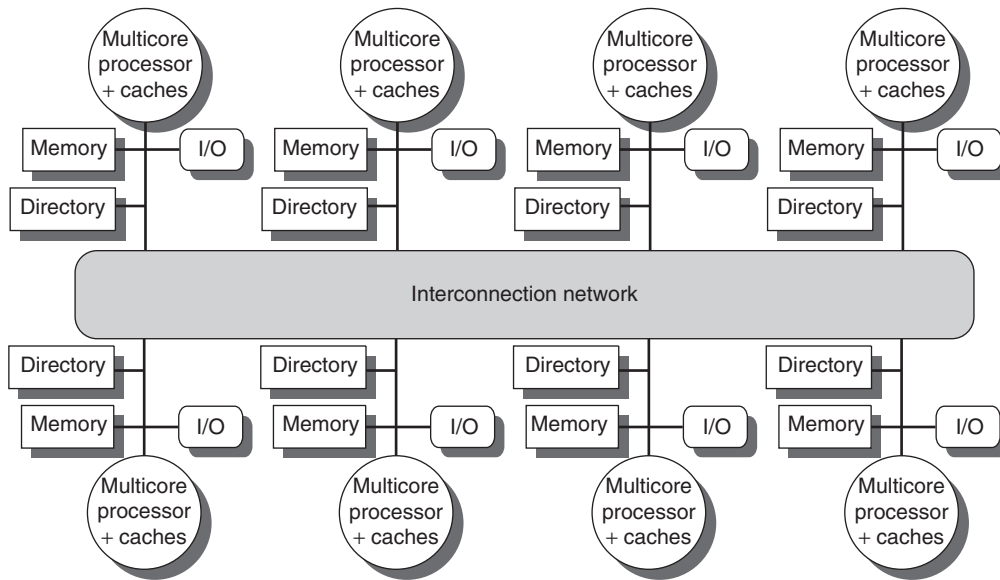
small, most of the traffic is coherence traffic, which is unaffected by cache size. Although a modern bus might accommodate 4 GB/sec, 170 GB/sec is far beyond the capability of any bus-based system. In the last few years, the development of multi-core processors forced all designers to shift to some form of distributed memory to support the bandwidth demands of the individual processors.

We can increase the memory bandwidth and interconnection bandwidth by distributing the memory, as shown in [Figure 5.2](#) on page 348; 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 snooping-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 (or collections of caches) have copies of the block, whether it is dirty, and so on. Within a multicore with a shared outermost cache (say, L3), it is easy to implement a directory scheme: Simply keep a bit vector of the size equal to the number of cores for each L3 block. The bit vector indicates which private caches may have copies of a block in L3, and invalidations are only sent to those caches. This works perfectly for a single multicore if L3 is inclusive, and this scheme is the one used in the Intel i7.

The solution of a single directory used in a multicore is not scalable, even though it avoids broadcast. The directory must be distributed, but the distribution must be done in a way that the coherence protocol knows where to find the directory information for any cached block of memory. The obvious solution is to distribute the directory along with the memory, so that different coherence requests 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, together with the maintenance of information that says what other nodes may be caching the block, is what allows the coherence protocol to avoid broadcast. [Figure 5.20](#) shows how our distributed-memory multiprocessor looks with the directories added to each node.

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 L2 or L3 cache block) times the number of nodes, where a node is a single multicore processor or a small collection of processors that implements coherence internally. This overhead is not a problem for multiprocessors with less than a few hundred processors (each of which might be a multicore) 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, but only supercomputer-sized systems need to worry about this.



**Figure 5.20** A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. In this case, a node is shown as a single multicore chip, and the directory information for the associated memory may reside either on or off the multicore. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism would handle both the maintenance of the directory information and any coherence actions needed within the multicore 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 nodes have the block cached, and the value in memory is up to date (as well as in all the caches).
- *Uncached*—No node has a copy of the cache block.
- *Modified*—Exactly one node has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.

In addition to tracking the state of each potentially shared memory block, we must track which nodes 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 chip (which is likely a multicore) 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 processes of invalidating and 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 the 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 5.21 shows the types 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

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node 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 5.21** The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local node to the home. The fourth through sixth messages are messages sent to a remote node 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.

**directory entry of an address reside.** The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The 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 5.6](#) when we discuss memory consistency models. In this section, we use this assumption to ensure that invalidates sent by a node 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 acknowledgments are required in response to write misses and invalidate requests. We discuss these issues in more detail in Appendix I.

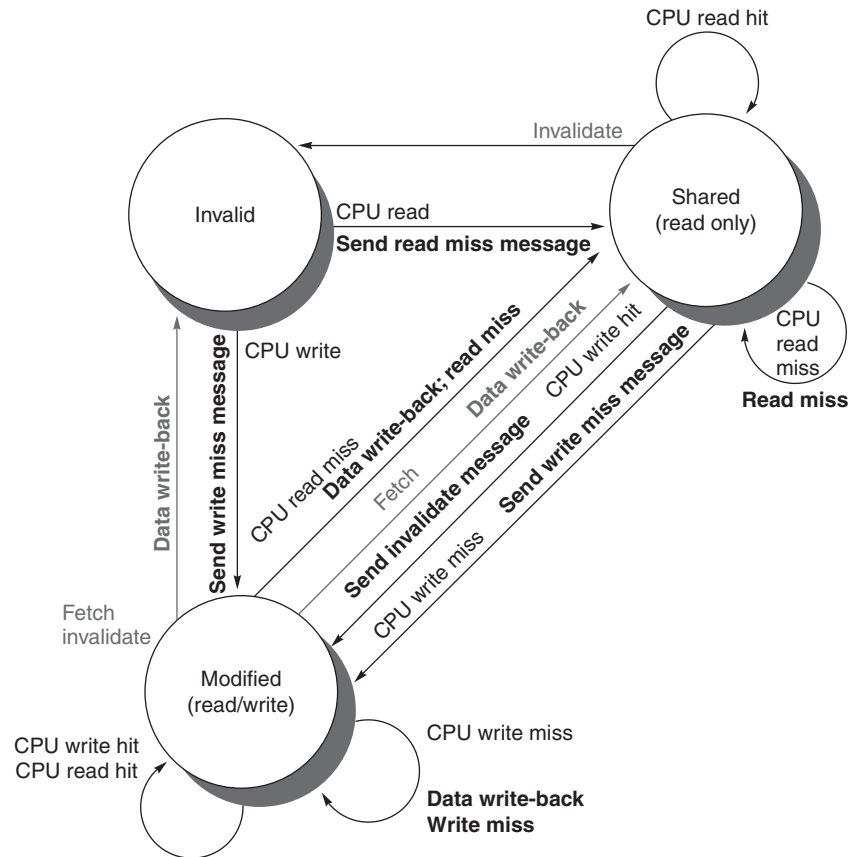
## An Example Directory Protocol

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 I.

[Figure 5.22](#) 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; [Figure 5.22](#) shows these operations. 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.



**Figure 5.22** 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.

The operation of the state transition diagram for a cache block in [Figure 5.22](#) 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 many multicore processors, the outermost level in the processor cache is shared among the cores (as is the L3 in the Intel i7, the AMD Opteron, and the IBM Power7), and hardware at that level maintains coherence among the private caches of each core on the same chip, using either an internal directory or snooping. Thus, the on-chip multicore coherence mechanism can be used to extend coherence among a larger set of processors by simply interfacing to the outermost shared cache. Because this interface is at L3, contention between the processor and coherence requests is less of an issue, and duplicating the tags could be avoided.

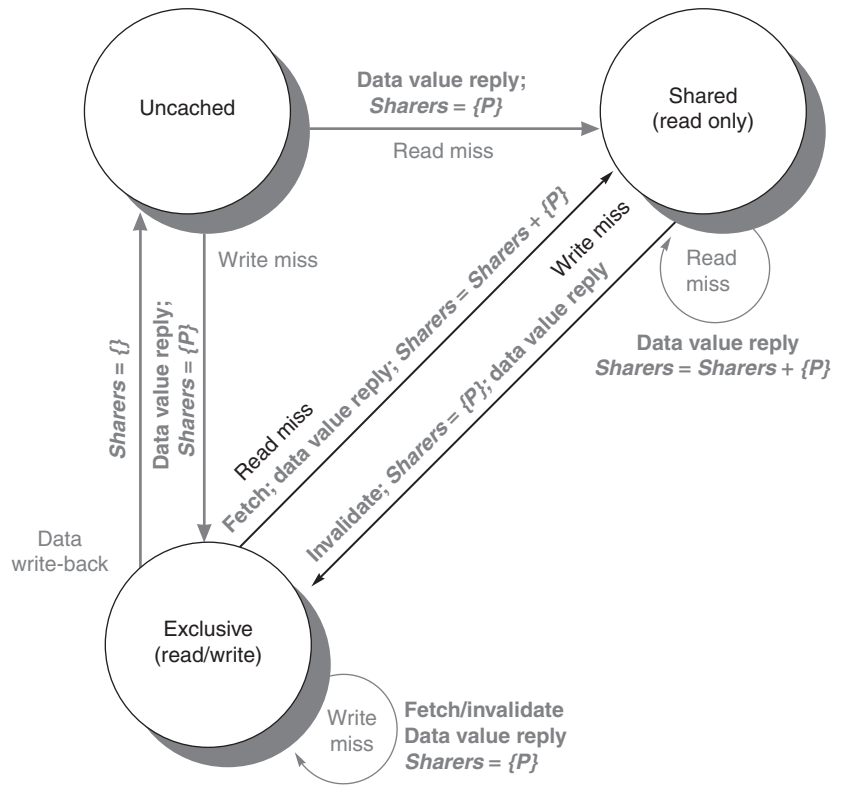
In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of actions: updating the directory state and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a block; unlike in a snooping 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 nodes that have a copy of a block; we use a set called *Sharers* to perform this function. In multiprocessors with fewer than 64 nodes (each of which may represent four to eight times as many processors), this set is typically kept as a bit vector. Directory requests need to update the set *Sharers* and also read the set to perform invalidations.

[Figure 5.23](#) 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.

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 node is sent the requested data from memory, and the requestor is made the only sharing node. The state of the block is made shared.



**Figure 5.23** 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.

- **Write miss**—The requesting node 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 node is sent the requested data from memory, and the requesting node is added to the sharing set.
- **Write miss**—The requesting node is sent the value. All nodes in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting node. 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 a cache on the node identified by the set Sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner 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 node 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 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 node, 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 5.23](#) 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 I 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 snooping protocols beginning on page 365. There are, however, new and additional problems, which we describe in Appendix I. In [Section 5.8](#), we briefly describe how modern multicores extend coherence beyond a single chip. The combinations of multichip coherence and multicore coherence include all four possibilities of snooping/snooping (AMD Opteron), snooping/directory, directory/snooping, and directory/directory!

## 5.5

## 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 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 I.

## 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). Anytime 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 be to keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say, atomic exchange from page 387, 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:  EXCHR2,0(R1)    ;atomic exchange
          BNEZR2,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:  LDR2,0(R1)      ;load of lock
        BNEZR2,lockit   ;not available-spin
        DADDUIR2,R0,#1  ;load locked value
        EXCHR2,0(R1)    ;swap
        BNEZR2,lockit   ;branch if lock wasn't 0
```

Let’s examine how this “spin lock” scheme uses the cache coherence mechanisms. Figure 5.24 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	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
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; state shared.
4		(Waits while bus/directory busy)	Lock = 0 test succeeds	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; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

**Figure 5.24** 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), and the value of the lock is 1 (i.e., locked); it is initially exclusive and owned by P0 before step 1 begins. P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3 to 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 take much longer. Once step 8 is reached, the process can repeat with P2, eventually getting exclusive access and setting the lock to 0.

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:  LLR2,0(R1)      ;load linked
        BNEZR2,lockit   ;not available-spin
        DADDUIR2,R0,#1  ;locked value
        SCR2,0(R1)      ;store
        BEQZR2,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.

## 5.6

**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 are really asking 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;          .....
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 now 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 5.7](#). 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 accesses 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  $X \rightarrow Y$ , meaning that operation  $X$  must complete before operation  $Y$  is done. Sequential consistency requires maintaining all four possible orderings:  $R \rightarrow W$ ,  $R \rightarrow R$ ,  $W \rightarrow R$ , and  $W \rightarrow W$ . The relaxed models are defined by which of these four sets of orderings they relax:

1. Relaxing the  $W \rightarrow R$  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  $W \rightarrow W$  ordering yields a model known as *partial store order*.
3. Relaxing the  $R \rightarrow W$  and  $R \rightarrow R$  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 [Section 5.7](#).

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. We then examine the performance gained when multithreading is added to multiprocessing.