5

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

Since 2004 processor designers have increased core counts to exploit Moore's Law scaling, rather than focusing on single-core performance. The failure of Dennard scaling, to which the shift to multicore parts is partially a response, may soon limit multicore scaling just as single-core scaling has been curtailed.

Hadi Esmaeilzadeh, et al., Power Limitations and Dark Silicon Challenge the Future of Multicore (2012)

Computer Architecture. https://doi.org/10.1016/B978-0-443-15406-5.00005-5

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, some of 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 has led 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 to increase performance faster than the basic technology allows (from a switching perspective) is through multiprocessing, including both SIMD and MIMD approaches.
- 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 and AI accelerators, which use other forms of parallelism), either because current performance is acceptable or because highly compute- and data-intensive applications are being done on the cloud.
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant inherent parallelism, arising from large datasets (usually in the form of data parallelism), "natural-world" parallelism (which occurs in scientific and engineering 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.

- The design of multicore processors with different types of cores (big/little), allowing systems to use high-performance (and higher-power) cores or more efficient low-power cores, as well as to incorporate domain-specific accelerators.
- Support for building systems consisting of multiple multicores by incorporating the interconnect and memory coherency support in the individual chips.

The third quotation reminds us that multicore may provide only limited possibilities for scaling performance. The combination of Amdahl's law effects and the end of Dennard scaling means that the future of multicore may be limited, at least as a method of scaling up the performance of single applications. Heterogeneous and domain-specific processors may offer other paths forward. We return to this topic later in the chapter.

In this chapter we focus on exploiting *thread-level parallelism* (*TLP*). TLP implies the existence of multiple program counters and thus is exploited primarily through MIMD (Multiple Instruction, Multiple Data) processors. Although MIMDs have been around for decades, the movement of TLP to the forefront across the range of computing from embedded applications to high-end servers is a product largely of the past 30 years. Likewise, the extensive use of TLP for a wide-range of general-purpose applications, versus either transaction processing or scientific applications, is more recent.

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 (possibly with replication) and that share memory through a shared address space. Such systems exploit TLP 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 an 8-core single-socket processor to dozens and sometimes hundreds of cores that communicate and coordinate primarily 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 *multicores*, and computers consisting of multiple chips, each of which is typically a multicore.

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 data centers typically built from very large numbers of processors (thousands to hundreds of thousands), connected with networking technology (not necessarily the same networking technology used to connect computers to the Internet); these large-scale systems are used for cloud computing primarily with massive numbers of independent tasks being executed in parallel. The multicores and multiprocessors we discuss in this chapter are building blocks for these larger-scale systems. In addition, computationally intensive tasks that can be easily made parallel, such as search and certain machine learning algorithms, have also made use of clusters. When these clusters grow to tens of thousands of servers and beyond, they are also called *warehouse-scale computers*. Alibaba, Amazon, Google, Microsoft, and Facebook all build and use warehouse-scale computers. These systems are sometimes also called multicomputers, particularly when used for scientific applications.

Many other books, such as Culler et al., (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 larger computers, as well as their use for computationally-intensive scientific applications.

Our focus will be on multiprocessors with roughly 8−1024 processor cores, which might occupy anywhere from 1 to 64 separate chips, possibly packaged in modules containing several CPUs as well as memory in one socket. Such designs vastly dominate in terms of both units and dollars. In large-scale multiprocessors the interconnection networks are a critical part of the design, and 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; with multithreading, which is present in most high-end multicore chips today, that number is 2−8 times higher. 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 TLP efficiently, the important qualitative distinction from ILP is that TLP is identified at a high level by the software system or programmer and that the threads consist of thousands to billions of instructions that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is usually higher than would be seen with a SIMD processor, such as a vector unit or 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.

Traditionally, shared-memory multiprocessors fell into two classes, depending on the number of processors involved, which in turn dictated 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 continues to change over time.

Early multiprocessors often used a single-shared memory and a bus connecting the processors and the memory. These multiprocessors are called UMA (Uniform Memory Access) multiprocessors because all processors have the same access time to memory. Some early multicores were also UMAs. The common bus simplified the design but also became a bottleneck as the number of cores and the performance of each core both grew. Thus most multicores changed to a structure like that in Figure 5.1, which employs a shared Last Level Cache

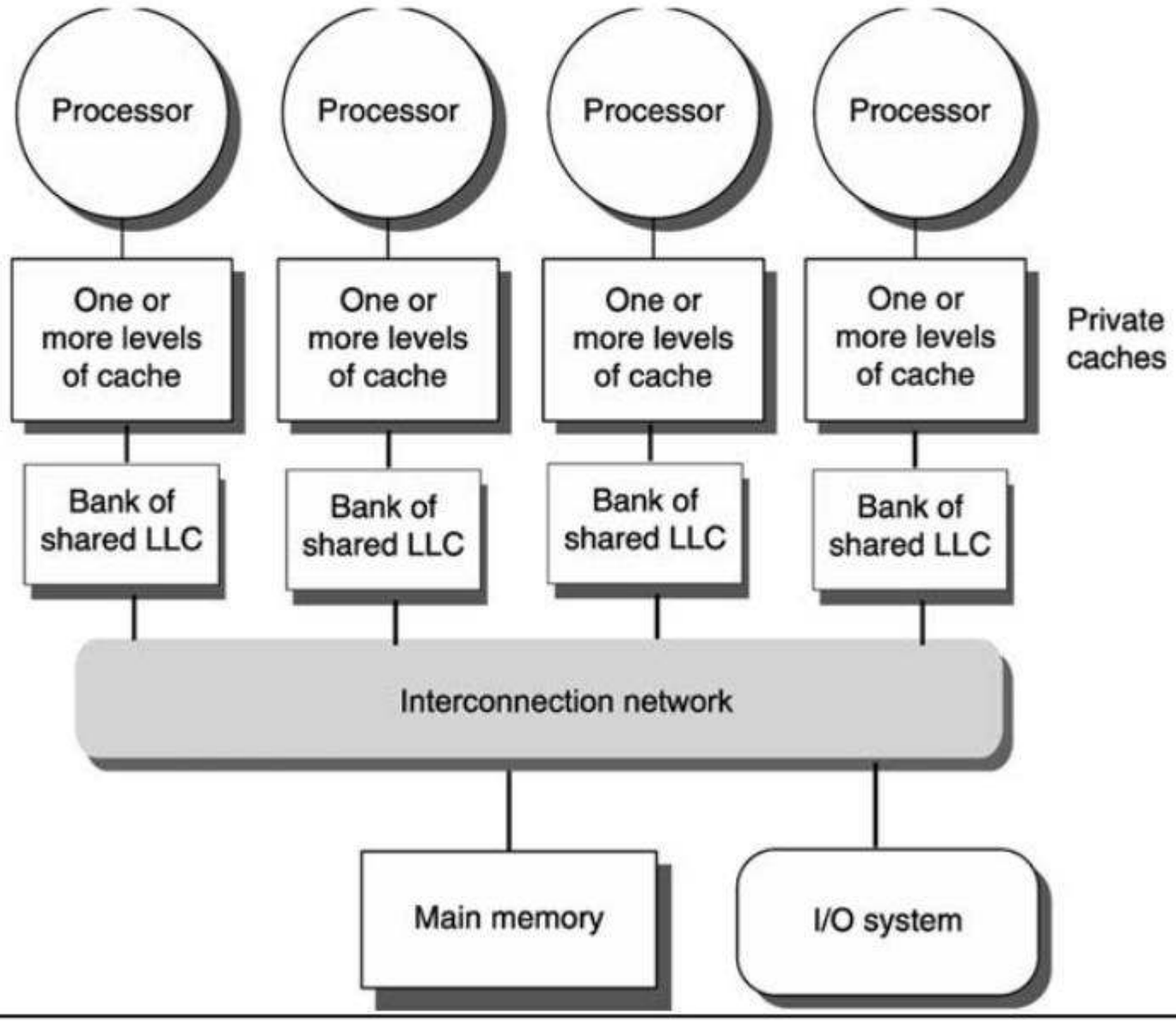


Figure 5.1 Basic structure of a multicore chip. Modern multiprocessors typically use 2 levels of private cache and a shared (sometimes noninclusive) L3. In most modern multicores the L3 is sliced into multiple banks with each bank associated with one or two multicores. The bus of earlier generation multiprocessors is replaced with an interconnection network, which for larger multicores is often a multistage, indirect network requiring multiple hops to get to some multicores. Memory is typically accessed through the same interconnection network, as are I/O devices. Note that even if the designs are UMA, they may be NUCA (NonUniform Cache Access), since the time to get to a shared bank of the LLC varies.

(LLC), which is typically distributed. An interconnection network allows the cores to access the memory interface, as well as permit communication among the LLCs, which we will see is needed to maintain a coherent view of memory. In some designs small groups of processors may share a slice of the LLC.

Multiprocessors consisting of multiple multicores may use both multiple memories and an interconnection network to connect the multicores, as Figure 5.2 shows. These architectures are called *distributed shared memory* (DSM) and exhibit Nonuniform Memory Access (NUMA) with the latency depending on the relationship between the core accessing the memory and the physical memory holding the data. Although the memory is physically distributed, the processors share a single-address space and can access all of the memory, as indicated by the name distributed *shared* memory. Processes and threads that need to communicate request the operating system to map some portion of their virtual address space to a shared portion of the physical address space. The memory system then ensures that the processors can access and share the memory correctly, a subject we will explore in depth in this chapter.

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

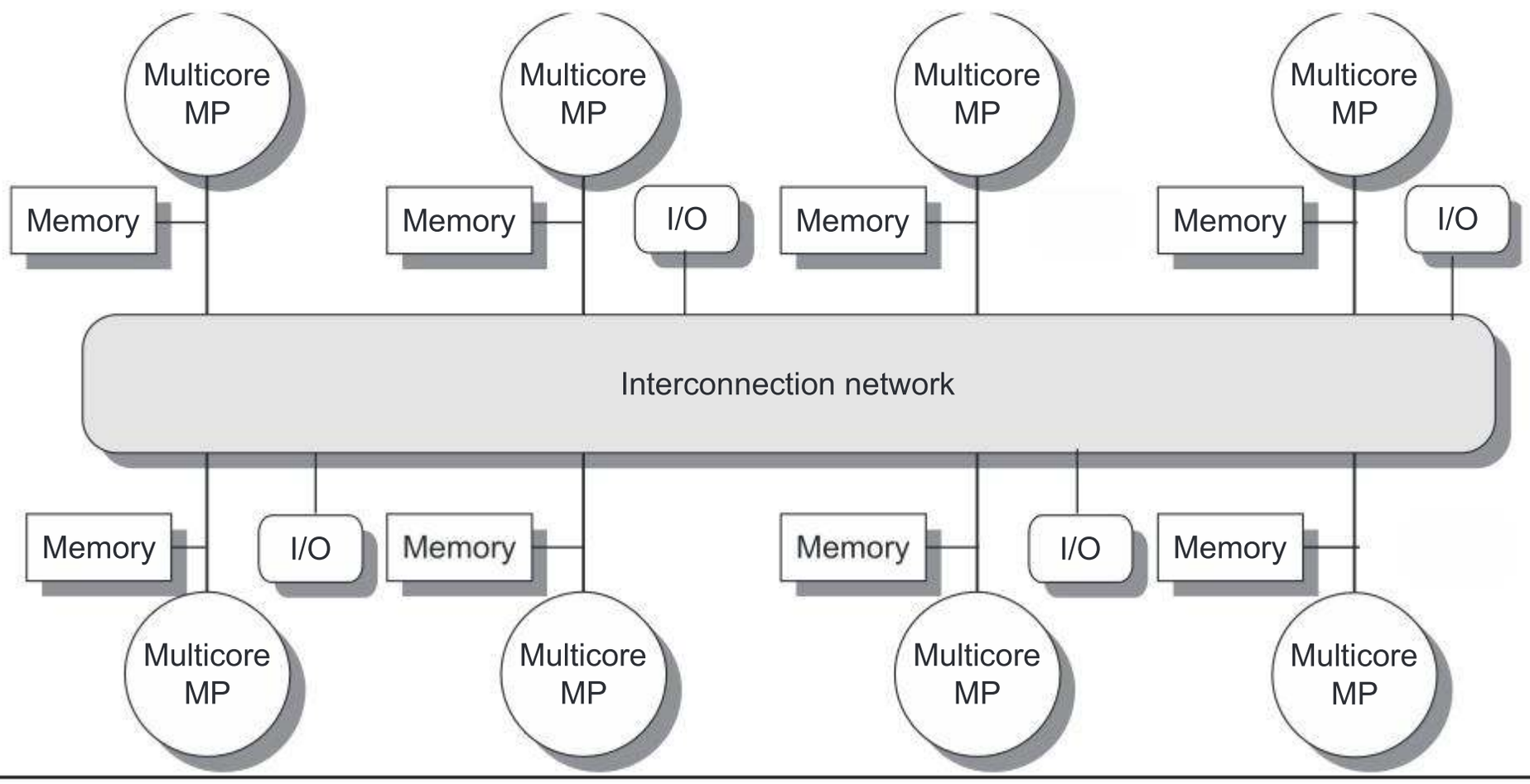


Figure 5.2 The basic architecture of a distributed-memory multiprocessor in 2023 typically consists of a multicore multiprocessor chip (with a shared LLC) with memory and an interface to an interconnection network that connects all the nodes, as well as I/O. Each processor core shares the entire memory, although the access time to the local memory attached to the core's chip will be much faster than the access time to remote memories. Many of these designs also have NUCA as well.

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 of dependent, interacting threads challenging. To overcome these hurdles typically requires a comprehensive approach that addresses the choice of algorithm and its implementation, the underlying programming language and system, the operating system and its support functions, and the architecture and hardware implementation. Although in many instances, one of these is a key bottleneck, when scaling to larger processor counts (say 32 or more), often *all* aspects of the software and hardware need attention.

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

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

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

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

Simplifying this equation yields:

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times \left(1 - \text{Fraction}_{\text{parallel}}\right) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

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. Amdahl's law can be used to analyze applications with varying amounts of speedup, as the next example shows.

Example Suppose we have an application running on a 100-processor multiprocessor, and assume that application can use 1, 50, or 100 processors. If we assume that 95% of the time we can use all 100 processors, how much of the remaining 5% of the execution time must employ 50 processors if we want a speedup of 80?

Answer We use Amdahl's law with more terms:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{100}}{\text{Speedup}_{100}} + \frac{\text{Fraction}_{50}}{\text{Speedup}_{50}} + (1 - \text{Fraction}_{100} - \text{Fraction}_{50})}$$

Substituting in:

$$80 = \frac{1}{\frac{0.95}{100} + \frac{\text{Fraction}_{50}}{50} + (1 - 0.95 - \text{Fraction}_{80})}$$

Simplifying:

$$0.76 + 1.6 \times \text{Fraction}_{50} + 4.0 - 80 \times \text{Fraction}_{50} = 1$$

$$4.76 - 78.4 \times \text{Fraction}_{50} = 1$$

$$\text{Fraction}_{50} = 0.048$$

If 95% of an application can use 100 processors perfectly, to get a speedup of 80, 4.8% of the remaining time must be spent using 50 processors and only 0.2% can be serial!

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 50–100+ clock cycles and among cores on separate chips anywhere from 500 clock cycles to as much as 1000 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 that has a 200 ns delay to handle a 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 obviously optimistic. Processors are stalled on a remote request, and the processor clock rate is 4 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

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{200 \text{ ns}}{0.25 \text{ ns}} = 800 \text{ cycles}$$

Therefore we can compute the CPI:

$$\begin{aligned}\text{CPI} &= 0.5 + 0.20\% \times 800\\ &= 2.6\end{aligned}$$

The multiprocessor with all local references is 2.6/0.5 = 5.2 times faster! In practice, the performance analysis is much more complex because 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 worse because contention caused by many references trying to use the global interconnect can lead to increased delays, or the access time might be better if memory were distributed, and the access was to the local memory.

This problem could have also been analyzed using Amdahl's law, an exercise we leave to the reader.

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 multithreading (discussed in Chapter 3) or by using prefetching (a topic we covered 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 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 and their application to scientific work.

5.2 Multiprocessor Cache Coherence

Because the view of memory held by two different processors is through their individual caches, the processors could end up seeing different values for the same memory location, as Figure 5.3 illustrates. 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—such as the LLC—and some levels are private (e.g., 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.

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!

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 three properties hold:

1. *Uniprocessor read coherency*: 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.
2. *Multiprocessor Read Coherency*: 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. *Multiprocessor Write Serialization*: 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 *y* and then *z* are written to a location, processors can never read the value of the location as *z* and then later read it as *y*.

Uniprocessor read coherency preserves program order—we expect this property to be true even in uniprocessors. Multiprocessor Read Coherency 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 Multiprocessor 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 to 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 short 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 force 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 because 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. Migration can also reduce energy since a local cache access is much more energy efficient than accessing off-chip memory.

Because the caches make a copy of the data item in the local cache, coherent caches also provide replication for shared data that are being read simultaneously. Replication reduces both latency of access and contention for a shared data item being read. Supporting this migration and replication is critical to performance in accessing shared data and also saves energy. 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.

Ensuring Coherence: Invalidate versus Update

There are two ways to maintain the coherence requirements. One method is to ensure that a processor has exclusive access to a data item before writing that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies before 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: because the write requires exclusive access, any copy held by the reading processor must be invalidated. Therefore, when the read occurs in a cache that had an old copy of the data, it misses in the cache and is forced to fetch a new copy of the data. For a write, we also require that the writing processor has exclusive access, preventing any other processor from being able to

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 status bit indicating ownership of a block. The ownership bit indicates that a block may be shared for reads, but only the owning processor can write the block, and that 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 caches 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; recall from Chapter 2 that a cache is inclusive if any location in a higher-level cache (L1 and L2 in this case) is also in L3. We return to the topic of inclusion in Section 5.7.

write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race (we'll see how we decide who wins shortly), causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which now contains the updated value. Therefore this protocol enforces write serialization.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *write broadcast* protocol. Because a write update protocol must broadcast all writes to shared cache lines, it consumes considerably more bandwidth. For this reason, all recent multiprocessors have opted to implement a write invalidate protocol as the default, and we will focus only on invalidate protocols for the rest of the chapter.

Cache Coherence Protocols

Key to implementing a cache coherence protocol is tracking the state of any shared data blocks. The state of a cache block is kept using status bits associated

with the block, similar to the valid and dirty bits kept in a uniprocessor cache. 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.* The directory is typically either at a shared LLC or at the memory interface corresponding to the memory where the block resides. For scalability reasons, most multiprocessors that consist of more than one chip use multiple distributed directories, often one per chip. Many multicores associate the directory information with the LLC. Since they distribute the LLC into banks, the directory within a single multicore is distributed! 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. Because the information is kept only locally, every coherence transaction must be broadcast to every cache. With a broadcast medium like a bus, each cache controller monitors or *snoops* on the medium to determine whether they have a copy of a block that is being requested.

Snooping protocols became popular with multiprocessors using single-core microprocessors and caches attached to a single shared memory by a bus. Today, most multicore designs use a hybrid protocol: snooping is used to keep the individual nonshared caches (L1 and L2) coherent, while a directory scheme is used at the LLC or memory interface to decide which processors must be snooped to ensure coherence. Some multicores reverse the order: they use snooping of a group of 2–4 sockets (connected on a broadcast medium) and a directory to designate which cores within each socket will need to be snooped. We will return to hybrid schemes at the end of section 5.4.

5.3 Maintaining Cache Coherence with Snooping

As we stated, snooping can be used to maintain coherence either for a single processor (and its private caches) or a small group of processors connected by a broadcast medium, which we will assume is a bus. We explain the protocol assuming a bus with multiple processors connected, but the basic snoop operations are exactly the same when there is a single processor and its caches.

The key advantage of the bus is that its broadcast ability makes it easy to perform the two critical operations in the coherence protocol: invalidation and locating the most current value of a shared data item. For example, 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 data item

corresponding to the address on the bus is in their cache. If so, the corresponding address in the cache is 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 would also serialize their writes. One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access. (We will return to this issue later, when we look at implementation details). All coherence schemes require some method of serializing accesses to the same cache block, either by serializing access to the communication medium or to another shared structure (such as the directory, covered in the next section).

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 because 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 because the most recent value of a data item can be in a private cache rather than in the shared cache or memory. Fortunately, 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 the LLC) 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 will typically take longer than retrieving it from the LLC. Because 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 L2 and L3 caches, 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 because they simply rely on the snooping capability. For writes, we want to know whether any other copies of the block are cached because, if there are no other cached copies, then the write does not need to 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 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 called the *owner* of the cache block.

After the 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. Because our snooping cache also sees all 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. If the LLC is inclusive (recall this means that any item contained in L1 or L2 is contained in the LLC), we can use the LLC to determine whether a snoop is required. In this fashion the inclusive LLC separates out misses that can go directly to memory and those that require a snoop. Many recent multicores enforce inclusivity for L2 (but not for the LLC); in such cases we can snoop L2 (which may have multiple banks to increase bandwidth, thus reducing contention) and only snoop L1 if L2 gets a hit.

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 as 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 (i.e., one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). In a processor with nonblocking caches such transactions can be overlapped. The MSHRs (Miss Status Handling Registers) can be used to determine what cache block needs to be updated to complete a transaction. 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, and we can also use the same protocol for maintaining the coherence of individual cores and connecting them to a directory mechanism for greater scalability.

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, whereas 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 two-thirds of the table) as well as those coming from the bus (in the bottom third of the table). 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 19.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or Exclusive	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	Exclusive	Replacement	Address conflict miss: write-back block; then place read miss on bus.
Write hit	Processor	Exclusive	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because 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	Exclusive	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	Exclusive	Coherence	Attempt to read shared data: place cache block on bus, write-back block, 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	Exclusive	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.

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,

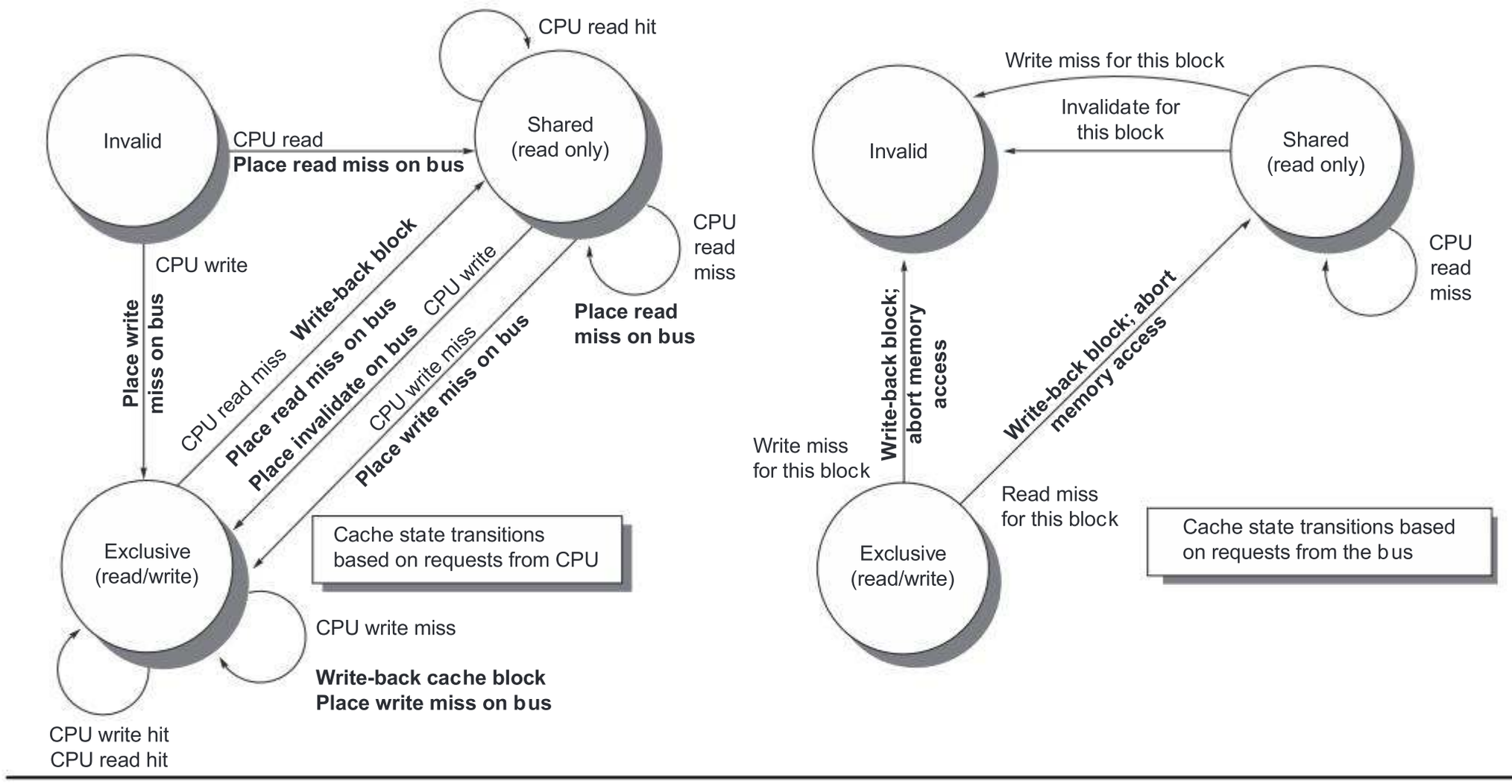


Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the local processor without a state transition shown in parentheses under the 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. Thus 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 a shared, inclusive LLC cache 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.

the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top portion of the table in Figure 5.5), as opposed to transitions based on bus requests (on the right, which corresponds to the bottom portion 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 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 either in a shared LLC or in memory, and this property simplifies the implementation.

Although our simple cache protocol is correct, it omits several 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, but this is not true, since many of the operations will take a number of clock cycles. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action, but this is not possible in a practical implementation. In fact, even a read miss might not be atomic; after detecting a miss in the L2 of a multicore, the core must arbitrate for access to the bus connecting to the shared L3 or memory. We will discuss how this difficulty is overcome shortly.

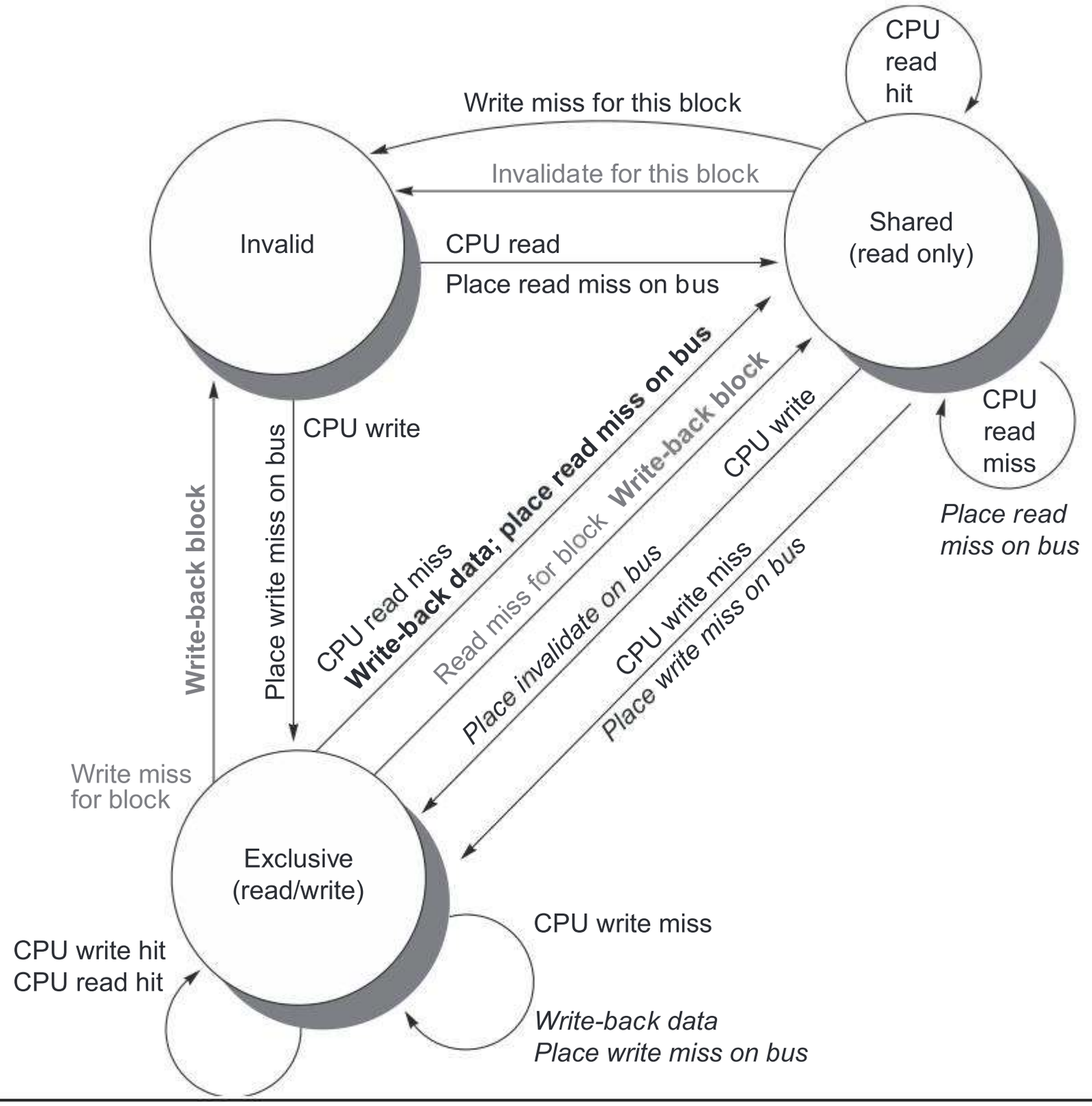


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in *black* and by the bus activities shown in *gray*. As in Figure 5.6, the activities on a transition are shown in *bold*.

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 an MSI (Modified, Shared, Invalid) protocol. There are many extensions of this basic protocol, which are created by adding additional states and transactions that optimize certain behaviors, possibly resulting in improved performance. Two of the most common extensions are:

1. *MESI*, which adds the state Exclusive to the basic MSI protocol, yielding four states (Modified, Exclusive, Shared, and Invalid). The exclusive state indicates that a cache block is resident in only 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. Once the block is written, the state is changed from Exclusive to Modified. If 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. This new state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a bock is modified. MESI is the protocol used by the CXL standard, an interconnect standard supporting coherent memory among processors and accelerators. The Intel i7 and i9 use a variant of a MESI protocol, called MESIF, which adds a state (Forward) to designate which sharing processor should respond to a miss and make the data available. It is designed to enhance performance in distributed memory organizations.

2. *MOESI*, which 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 its value 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 processor family uses the MOESI protocol. More recent AMD processors have used a protocol based on MOESI but including both the F state mentioned previously and a D (Dirty) state. The D state is a variant on the Owned state: indicating that the copy in memory is out of date.

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. For multicores, a single shared bus became a bottleneck with only a few cores. As a result, multicore designs have gone to higher bandwidth interconnection schemes, as well as multiple, independent memories to allow larger numbers of cores. The multicore chips we examine in Section 5.8 use three different approaches:

1. The IBM Power 10 processor, which has up to 15 cores in a single chip, uses 16 parallel rings that connect the distributed L3 caches and up to 16 separate memory channels.

2. The Xeon Platinum series comes as a single chip with up to 40 cores or a chiplet with four processor chips and HBM memory or cache. It uses multiple rings to connect up to 40 cores on a chip, a distributed 60 MiB LLC cache, and up to eight memory channels (depending on the configuration). Sapphire rapids, the newest Xeon server, has announced 56 cores and 4 chips per chiplet for a total of 224 cores.
3. The AMD EPYC is a chiplet-based design with 8 cores per chiplet and up to 4 chiplets per socket, and up to two sockets for 64 cores total. The LLC is shared on each chiplet, which also has its own memory channels. Infinity Path is used to connect the cores within and across chiplets. The next generation will support up to 12 chiplets for 96 cores in total.

In addition to the contention for the shared interconnect network due to cache misses and coherency traffic, snooping bandwidth at the caches can also become a problem because every cache must potentially examine every miss. Hence, having additional interconnection bandwidth might only push the contention to the cache. To understand this problem, consider the following example.

Example Consider a 16-processor multicore where each processor has its own L1 and L2 caches, and snooping is performed on a shared bus at the L2 caches. Assume the average L2 request, whether for a coherence miss or other miss, is 15 cycles. Assume a clock rate of 4.0 GHz, a CPI of 0.5, and a load/store frequency of 40%. If our goal is that no more than 50% of the L2 bandwidth is consumed by coherence traffic, what is the maximum coherence miss rate (CMR) per processor?

Answer Start with an equation for the number of cache cycles that can be used (where CMR is the coherence miss rate):

$$\text{Cache cycles available} = \frac{\text{Clock rate}}{\text{Cycles per request}} = \frac{4.0\ \text{Ghz}}{15} = 0.27 \times 10^9$$

$$\begin{aligned}\text{Cache cycles used} &= \text{Memory references/clock/processor} \times \text{Clock rate} \\ &\quad \times \text{processor count} \times 2 \times \text{CMR} \\ &= \frac{0.4}{0.5} \times 4.0\ \text{GHz} \times 16 \times 2 \times \text{CMR} = 102 \times 10^9 \times \text{CMR}\end{aligned}$$

$$\text{CMR} = \frac{0.27}{102} = 0.0026 = 0.26\%$$

This means that the CMR must be 0.26% or less. In the next section and Appendix I we will see several applications with CMRs in excess of 1%. Alternatively, if we assume that CMR can be 1%, then we could support only 4 processors. Clearly, even small multicores will require a method for scaling snoop bandwidth.

There are several techniques for increasing the snoop bandwidth:

1. The tags can be duplicated. This doubles the effective cache-level snoop bandwidth but becomes expensive with very large caches, and if the duplicate tags indicate a hit on a snoop, the primary tags will still need to be accessed. If the L2 has multiple banks, which many recent designs do, this effectively increases the snoop bandwidth.
2. If the LLC cache on a multicore is shared and inclusive then we can direct the snoop first to the LLC and only interrogate the L2 caches if there is a hit in the LLC (otherwise, we know the data cannot be present). By distributing the LLC into multiple banks, we can also increase the LLC bandwidth. This approach, used by the IBM Power 10, leads to a NUCA (NonUniform Cache Access) design but effectively scales the snoop bandwidth at the LLC by the number of banks. If there is a snoop hit in the LLC, then we must still broadcast to all L2 caches, which must in turn snoop their contents. Note that the LLC must be inclusive for this to work. As we will see, the Power 10 employs a sophisticated coherence scheme (inherited from early Power processors) that attempts to minimize broadcast snoops.

Directory schemes provide another method of scaling coherence bandwidth by avoiding snooping and the broadcast. Because directory schemes do not rely on a shared broadcast mechanism to serialize operations, they must use a request-response mechanism to know when a coherency operation has completed. Section 5.4 describes how directory-based schemes work.

The AMD Opteron represents another intermediate point in the spectrum between a snooping and a directory protocol. Memory is directly connected to each multicore socket, and up to four multicore sockets can be connected. The system is NUMA (NonUniform Memory Access) because 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 find potentially shared copies, like a snooping protocol, but uses the acknowledgments to order operations, like a directory protocol. Because local memory is only slightly faster than remote memory in the Opteron implementation, some software treats the Opteron multiprocessor as having uniform memory access.

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 MiB/s, which would not be enough bus bandwidth to support even one core of an Intel i9 in 2023! 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 2024 *every* multicore system with more than four cores uses an indirect on-chip interconnect (i.e., neither a bus or crossbar) within the package, and that indirect network is extended to allow for multiple chips to be connected in a cache-coherent multiprocessor. Thus designers must face the challenge of implementing snooping (or a directory scheme) without the simplification of a bus to serialize events, even within a single chip.

Implementing the coherence protocol shown earlier requires two properties:

1. Atomicity: coherence events (such as obtaining exclusive access) must appear atomic even though they take multiple clock cycles to occur. An event appears atomic if no other intervening event can change the outcome.
2. Serializability: coherence events must appear to occur in some serial order that all processors see as the same order.

If either of these properties did not hold, two processors could see different values for data items, indicating that the system was not coherent.

Serializability is easier to achieve. The coherence protocol must guarantee that all accesses to a particular address pass through one common point, which could be a bus, a shared LLC, or a memory module. This common point forces possible conflicting references to be ordered, thus ensuring that such references are serialized. The protocol should also handle all pending requests from the bus (or other communication network) before initiating a new transaction on behalf of the CPU.

The bigger difficulty in actually implementing the snooping coherence protocol is that write and upgrade misses are not naturally 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 finally updating the cache cannot be done as though they took a single cycle.

In a multicore with a single bus these steps are effectively made 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 early designs a single status line was used to signal when all necessary invalidates had been received and were being processed. Following that signal, the processor that generated the miss could release the bus, knowing that any required actions would be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively made the

individual steps atomic. Note that such an approach is simply not possible without the shared bus; furthermore, it is incompatible with a nonblocking cache!

Instead, a processor in the middle of handling a read or write miss (including an upgrade miss) must be able to accept an incoming transaction, invalidate the block, and restart the read or write request.

The simplest way to think about and implement the protocol is to add extra transient states to the protocol. The extra "hidden" states are placed within each transaction by the core that requires communication. These extra states are transient in the sense that they are waiting for the completion of an action in the memory system rather than a new request by the core. If, when in the transient state, a write miss or invalidate is received from the system (generated by another core), the state of the cache block is changed to invalid, and the core retries the access (which is now guaranteed to be a miss). It is easier to understand the role and operation of the hidden states in a table, as shown in Figure 5.8. These transient states effectively make the protocol actions atomic, by restarting any action that is interrupted by another action.

Performance of a Shared-Memory Multiprocessor

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 coherency transactions, which results in invalidations and subsequent

Cache state	Core action	Wait for	State when successful
Invalid	Read miss	Miss to return	Shared
Invalid	Write miss	Miss to return	Exclusive
Shared	Read miss	Miss to return	Shared
Shared	Write hit	Invalidate completion	Exclusive
Shared	Write miss	Miss to return	Exclusive
Exclusive	Read miss	Miss to return	Shared
Exclusive	Write miss	Miss to return	Exclusive

Figure 5.8 The Hidden, Transient States and Actions. This table shows the seven additional states needed. These states are transient: when the transaction listed under the column "Wait for" completes, the state is changed to the successful state, as would occur in the basic coherence protocol. If, however, a write miss or invalidate for the same block is received before the pending transaction completes, that block's state is changed to invalid, and the transaction must be restarted from that state. At that point, only a read miss or write miss can occur, since the block is invalid. Note that the write-backs shown in Figures 5.6 and 5.7 must also be done. Doing the write backs first is simpler; if they are buffered, then incoming transactions must also snoop on the buffer contents.

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 *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 because 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 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 (but in the same cache block) 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 that block is being 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 z1 and z2 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 z1	
2		Read z2
3	Write z1	
4		Write z2
5	Read z2	

Answer

1. Here are the classifications by time step: This event is a true sharing miss, since z1 is in the shared state in P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since z2 was invalidated by the write of z1 in P1, but that value of z1 is not used in P2.
3. This event is a false sharing miss, since the block containing z1 is marked shared due to the read in P2, but P2 did not read z1. The cache block containing z1 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 on a Small-Scale Multiprocessor

In this section we examine the memory system behavior of a 4-processor shared-memory multiprocessor when running an online transaction processing workload. The study we examine was done with a 4-processor Alpha system in 1998, but it remains the most comprehensive and insightful study of the performance of a multiprocessor for such workloads. We will focus on understanding the multiprocessor cache activity, and particularly the behavior in L3, where much of the traffic is coherence related.

The results were collected either on an AlphaServer 4100 or using a configurable simulator modeled after the AlphaServer 4100 (see Barroso et al. [1998]). 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 recently, 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 i9 and other processors, as shown in Figure 5.9, although the caches are smaller than those in the newer multicores. Rather than focus on the performance details, we consider data that looks at the simulated L3 behavior for L3 caches varying from 2 to 8 MiB per processor.

Although the original study considered three different workloads, we focus our attention on the online transaction-processing (OLTP) workload modeled after TPC-B (which has memory behavior like 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. Overall, 71% of the execution time is spent in user mode, 18% in the operating system, and 11% idle, primarily waiting for I/O. Of the commercial applications studied, the OLTP application stresses the memory system the hardest and shows significant challenges even when evaluated with much larger L3 caches. For example, on the AlphaServer, the processors are stalled for approximately 90% of the cycles, with memory accesses for L3 misses occupying almost half the stall time and L2 misses 25% of the stall time.

We start by examining the effect of varying the size of the L3 cache. In these studies the L3 cache is varied from 1 to 8 MiB per processor; at 2 MiB per processor, the total size of L3 is equal to that of the Intel i7 6700. In the case of the i7, however, the cache is shared, which provides both some advantages and disadvantages. It is unlikely that the shared 8 MiB cache will outperform separate L3

Cache level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I/8 KB D	32 KB I/32 KB D
	Associativity	Direct-mapped	8-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 MiB (total 8 MiB unshared)	2 MiB per core (8 MiB total shared)
	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. Both systems have a high penalty (125 cycles or more) for a transfer required from a private cache. A key difference is that L3 is shared in the i7 versus four separate, unshared caches in the Alpha server.

caches with a total size of 16 MiB. Figure 5.10 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 because of the reduction in L3 misses. Surprisingly, almost all the gain occurs in going from 1 to 2 MiB (or 4—8 MiB of total cache for the four processors). There is little additional gain beyond that, despite the fact that cache misses are still a cause of significant performance loss with 2 MiB and 4 MiB 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 5.11 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 MiB L3 are instruction and capacity/conflict misses. With a larger L3, these two sources shrink to be minor contributors. Unfortunately, the compulsory, false sharing, and true sharing misses are unaffected by a larger L3. Thus, at 4 and 8 MiB, 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 MiB.

Increasing the cache size eliminates most of the uniprocessor misses while leaving the multiprocessor misses untouched. How does increasing the processor

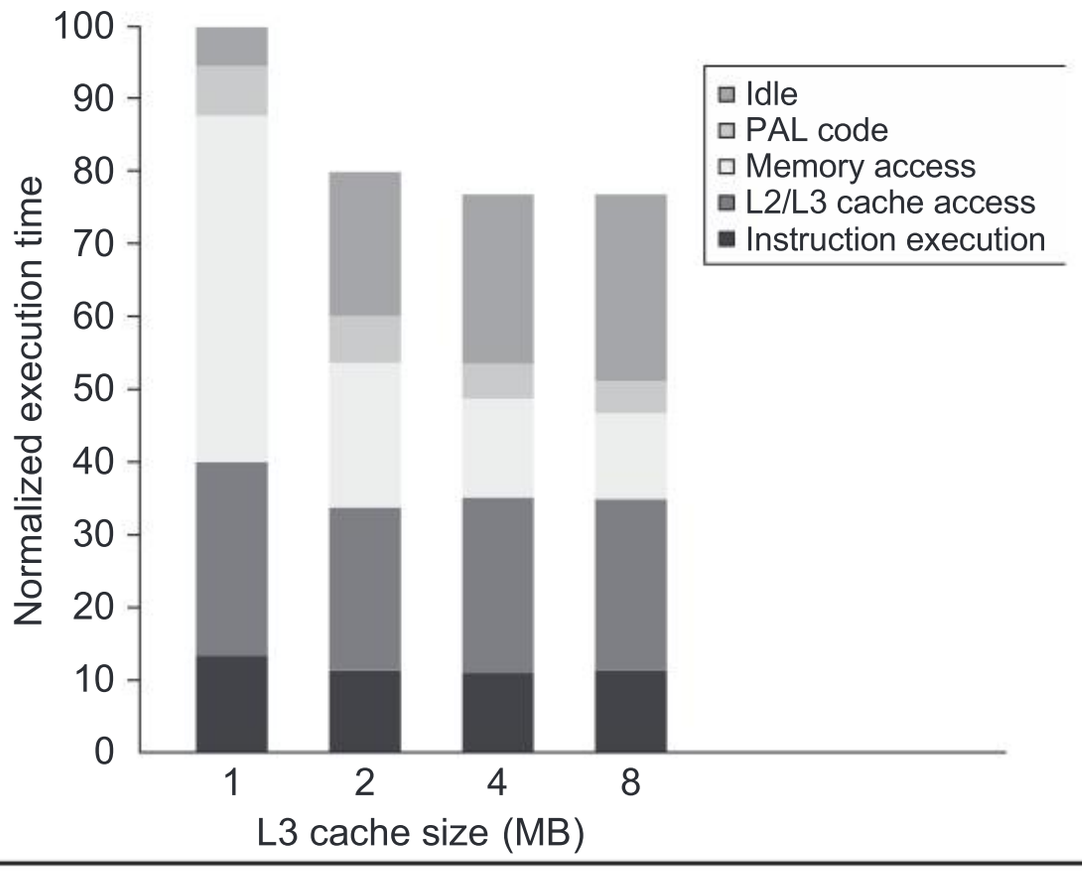


Figure 5.10 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 to 8 MiB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system stalls, 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.

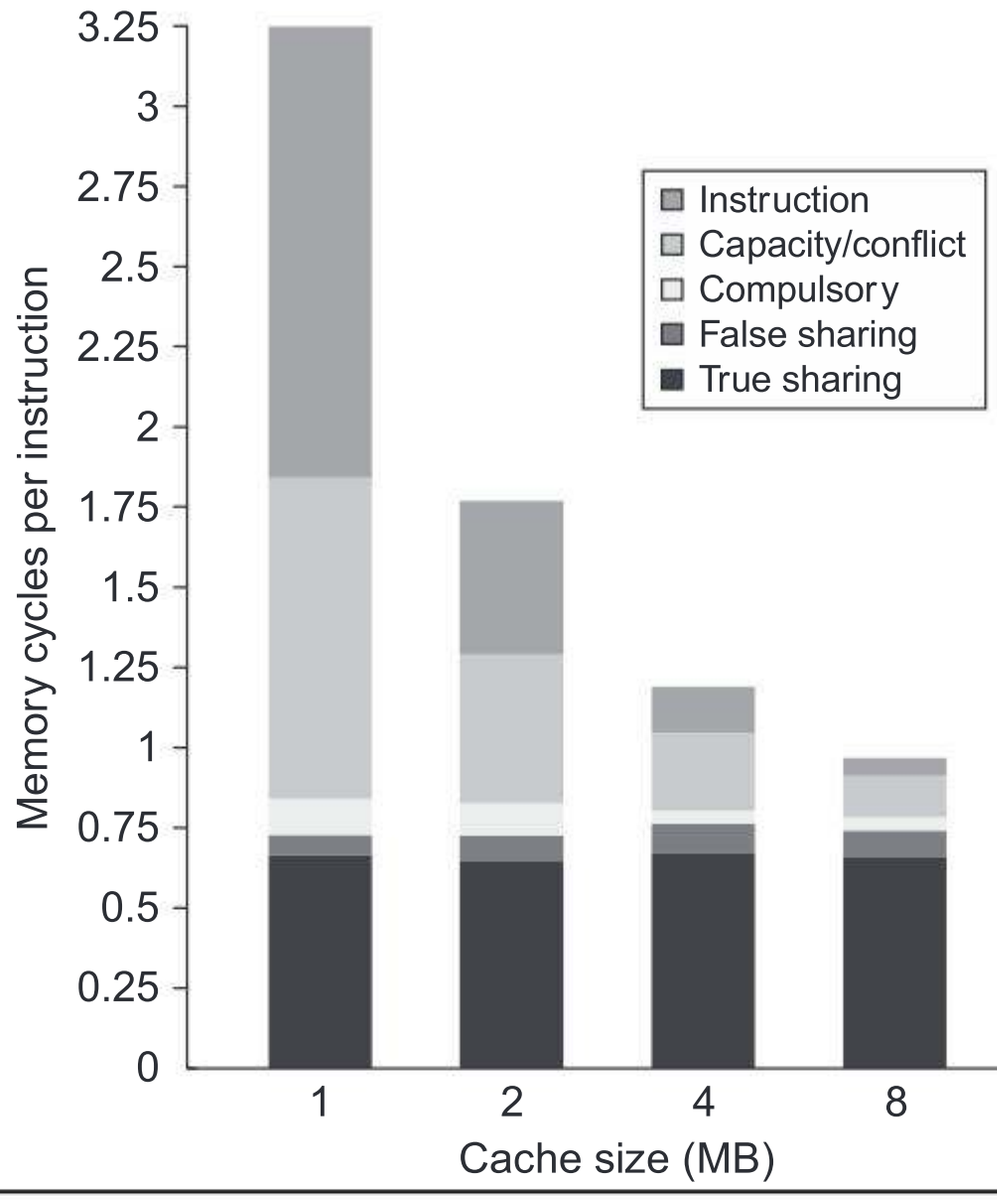


Figure 5.11 The contributing causes of memory access cycle shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

count affect different types of misses? Figure 5.12 shows these data assuming a base configuration with a 2 MiB, two-way set associative L3 cache (the same effective per processor cache size as the i7 but with less associativity). 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 5.13 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.

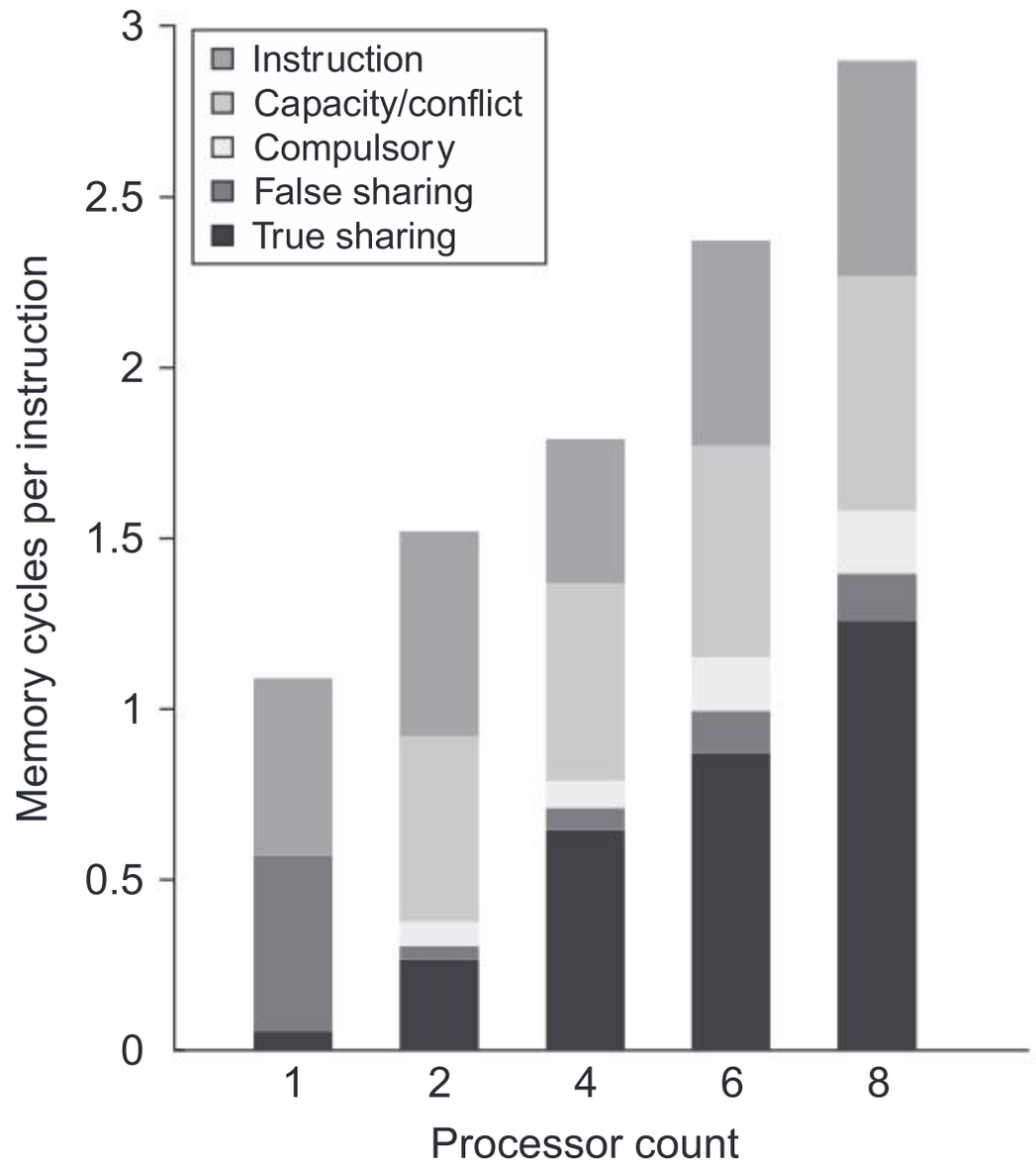


Figure 5.12 The contribution to memory access cycles increases as processor count increases primarily because of increased true sharing. The compulsory misses slightly increase because each processor must now handle more compulsory misses.

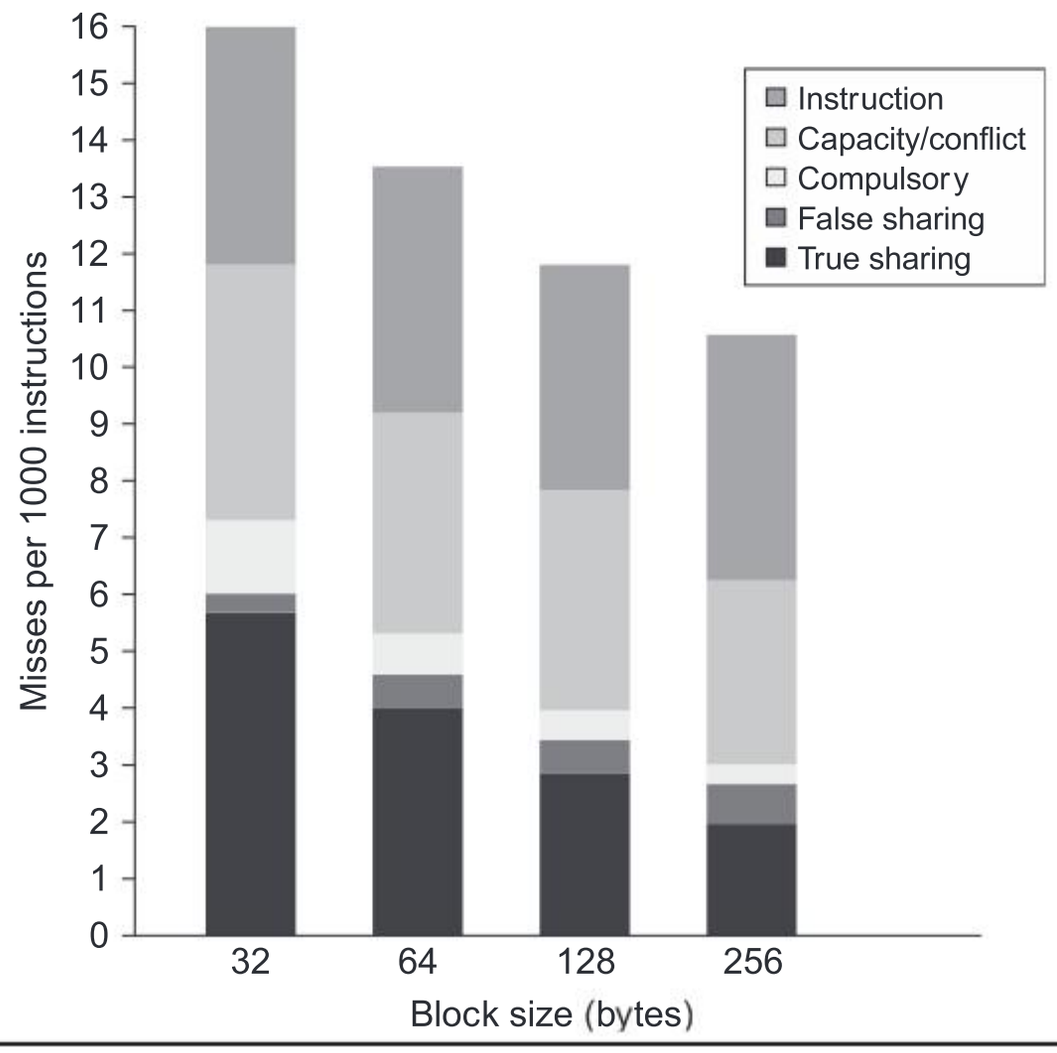


Figure 5.13 The number of misses per 1000 instructions drops steadily as the block size of the L3 cache is increased, making a good case for an L3 block size of at least 128 bytes. The L3 cache is 2 MiB, two-way set associative.

- 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 MiB.
- 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 mixed L2 and L3 caches, 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 are attainable, especially at the 64-byte (the i7/i9 block size) and 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. In Section 5.4 we will look at the performance characteristics of larger-scale, NUMA-based multiprocessors for a highly scalable, but loosely coupled, workload.

5.4 Maintaining Cache Coherence with Directories

As we saw in Section 5.3, 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.

We can increase the memory bandwidth and interconnection bandwidth by distributing the memory, as shown in Figure 5.2 on page XYZ; 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. This state includes which caches (or collections of caches) may have copies of the block and whether the block is modified.

For a single multicore with an inclusive shared LLC, it is easy to implement a directory at the LLC: simply keep a bit vector of the size equal to the number of cores for each LLC block. The bit vector indicates which private L2 caches may have copies of a block (since all L2 blocks must be in the LLC), and invalidations are sent only to those caches. This solution works perfectly for a single multicore if the LLC is inclusive, and this scheme is the one used in the Intel i7. Suppose the LLC is not inclusive; we can still store the directory information at the LLC, keeping either a separate directory structure (the Intel Xeon solution) or a copy of the L2 tags (the AMD EPYC solution). Having a single directory also simplifies the ordering of events, which is required to implement various models of memory consistency (see Section 6.6). With a single directory, we can simply hold back subsequent requests until we know that a currently operating request (e.g., an invalidation or fetch of remotely cached data) completes. This blocking approach is inefficient as the number of cores grows and will not work when there is more than one directory.

Instead of a single directory, the directory, like the memory, 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 5.14 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 LLC block) times the number of nodes, where a node is a single multicore processor or a small collection of processors that implements coherence internally by snooping. 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.

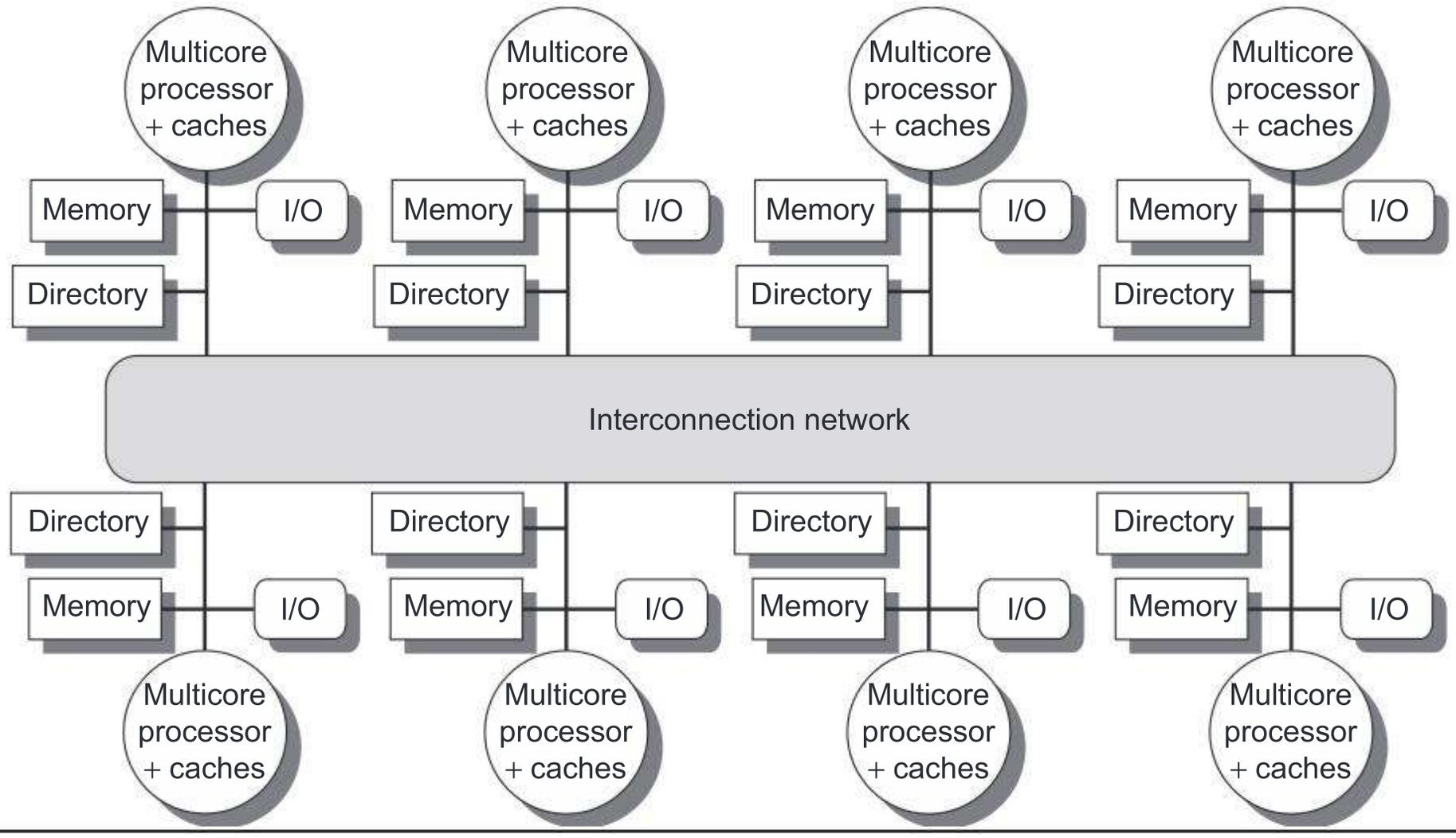


Figure 5.14 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. In this case a node is shown as a single multicore chip, and the directory information for the associated memory may reside either on or off the multicore. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism will 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 that have copies).
- *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 because 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 that might be cached. When the block is shared, each bit of the vector indicates whether the corresponding core (or possibly group of cores) 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 because 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 using a broadcast to all the nodes. Instead of broadcasting to all the nodes, directory-based systems use point-to-point messages between requesting cores, the directory controller, and responding cores.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the cores and the directories for the purpose of handling misses and maintaining coherence. Figure 5.15 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 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, whereas 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 because 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 assume that messages will be received and acted upon in the same order they are sent. This assumption, while true in the multiprocessors and multicores we examine in Section 5.8, may not be true in very large multiprocessors. When this strict ordering is not followed there are 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. Most of these issues are handled by adding transient states to accommodate nonatomic actions, just as we did in the snooping case.

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.
Acknow-ledge	Home directory	Local cache	A	Indicates all the invalidates requested for address A have been completed (the directory tracks these ars they arrive and sends the message when all shared copies have been invalida-ted)
Acknow-ledge	Remote cache	Home directory	P.A	Indicates that processor P has completed an invalidation for address A. The directory tracks these as they arrive before sending an acknowledgement to he local requesting cache

Figure 5.15 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 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 because any dirty block must be exclusive, and any shared block is always available in the home memory. The last two rows are acknowledgments needed to track when a remote cache has completed an invalidation, and when all the invalidations needed for a local cache to complete a write have been done. Because invalidations do not otherwise require a response and the consistency model requires that we know when a write can proceed, the acknowledgment messages are needed.

One critical addition, which we include as a message type in Figure 5.15, arises from the fact that we cannot know when all necessary invalidations are completed. As a result, when we send invalidations, which do not need to respond with any data, we must send an acknowledgment message to know when the invalidate has been completed. Furthermore, we cannot allow a write to proceed until we know that all invalidations have been completed. We will see why we need this restriction when we discuss memory consistency in Section 5.6, and we also 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. In practice, 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 5.16 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 5.16 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; transient states are required on these transactions, just as they were in the snooping case. We will also need to track invalidates at the directory to know when they are all completed.

The operation of the state transition diagram for a cache block in Figure 5.16 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 and i9, the AMD Opteron and EPYC, and the IBM Power series), 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. This design allows the directory information to be stored at the LLC. Extending beyond one socket, where the LLCs are no longer shared, requires some extensions, which we discuss shortly.

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

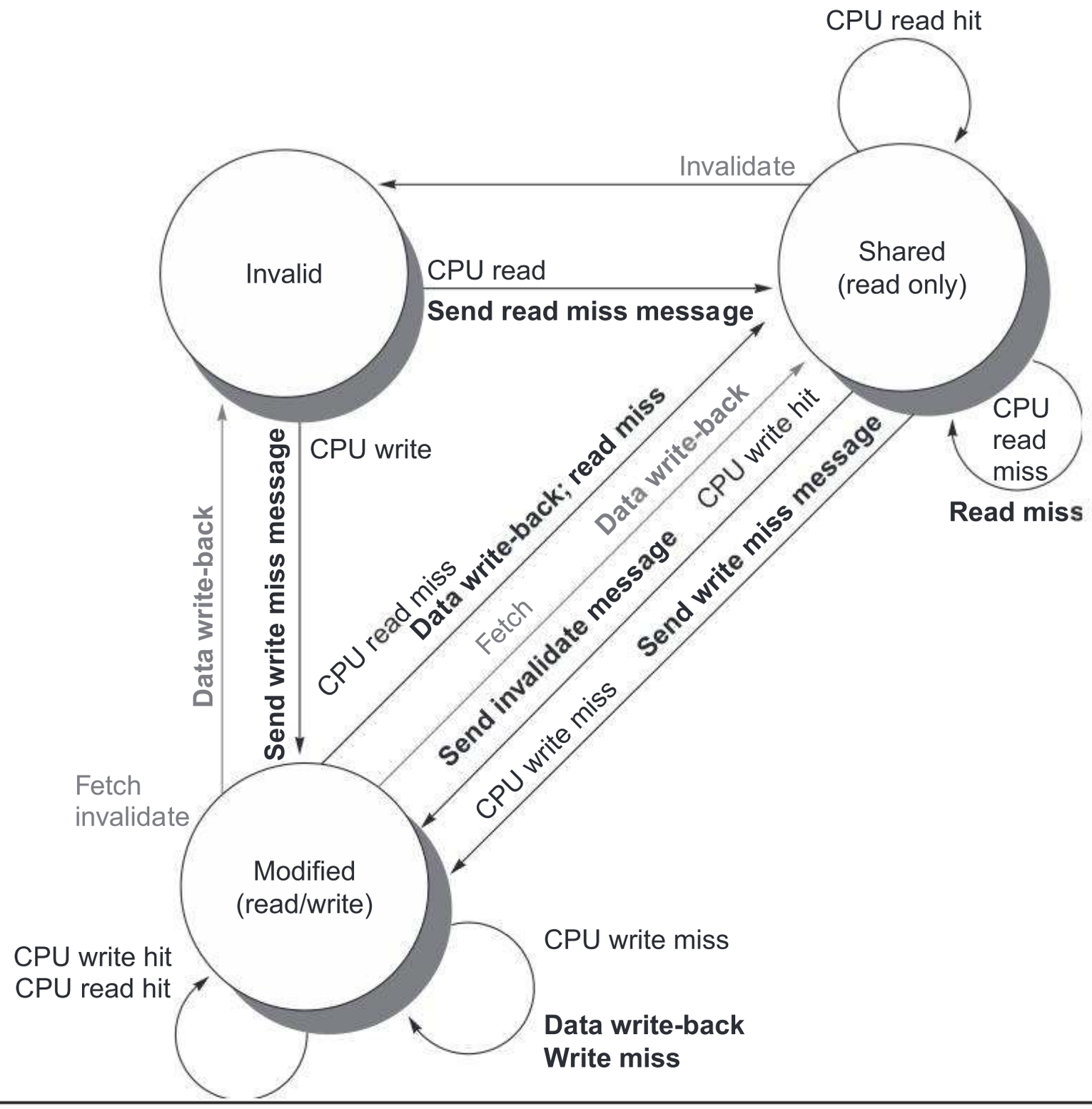


Figure 5.16 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in *black*, and those from the home directory are shown in *gray*. The states are identical to those in the snooping case, and the transactions are very similar, with 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.

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 read the set to perform invalidations.

Figure 5.17 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, and 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

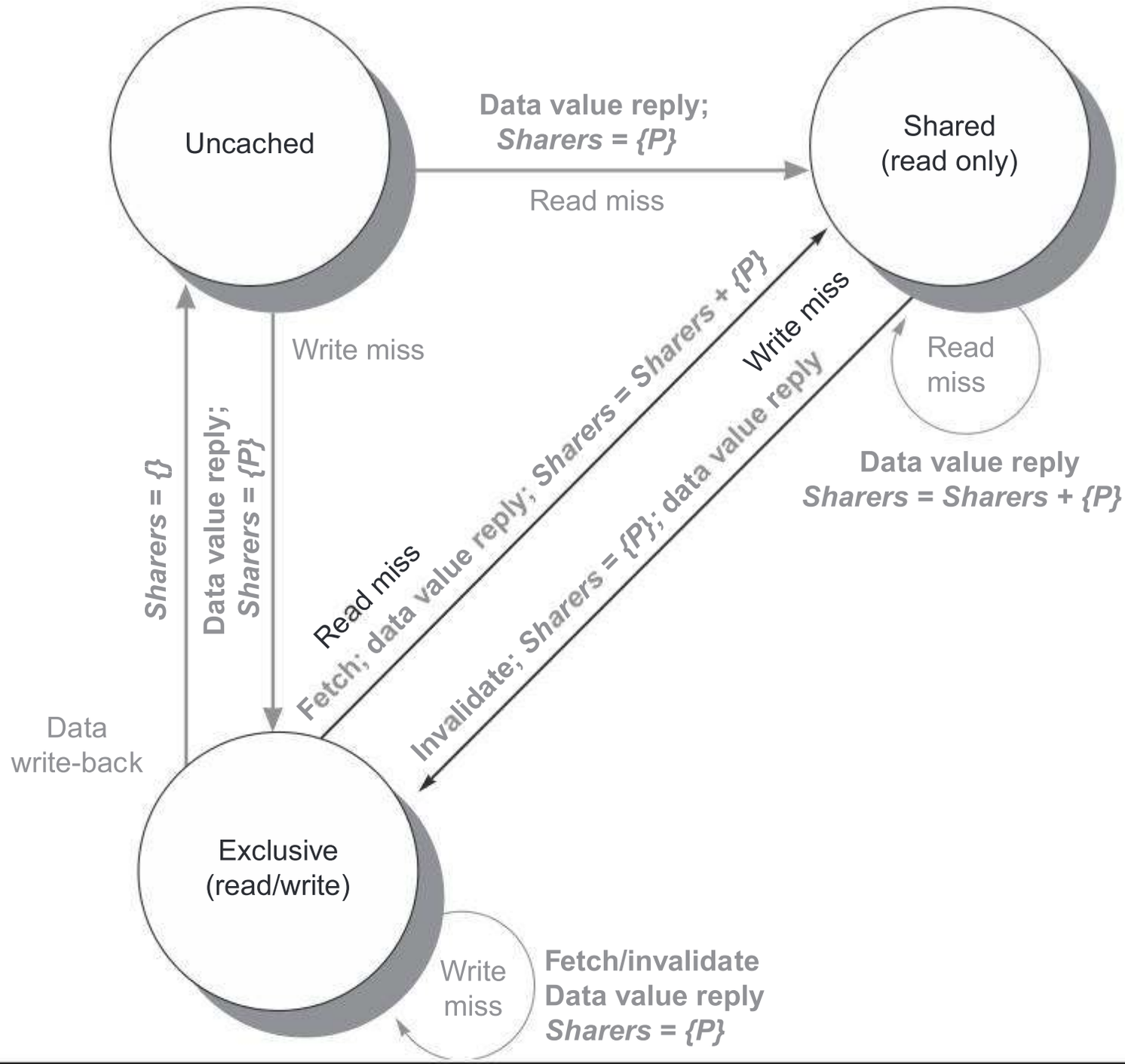


Figure 5.17 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. *Bold* indicates the action taken by the directory in response to the request. Just as in snoopy coherence, the directory cannot respond to a write miss until it knows that all invalidates have been seen. In a directory scheme this is tracked by acknowledgments to the directory from caches that have been sent invalidates.

sending it to another node; a realistic implementation cannot use this assumption and will require transient states like those we saw for the snooping scheme.

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 requester is made the only sharing node. The state of the block is made shared.
- *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 directory knows all invalidates have been completed.

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.

The directory protocols used in real multiprocessors have additional optimizations. For example, 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 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 add complexity by increasing the types of messages that must be handled. In addition, the same extensions that we described for snooping protocols (such as the addition of the O and I states) are also possible for a directory protocol.

Implementing a directory scheme requires solving most of the same challenges we discussed for snooping protocols. There are, however, additional problems, if the number of CPUs is not limited (see Appendix I).

Implementing Directory Coherence in a Multicore

All multichip multiprocessors have chosen to use a hybrid of snooping and some sort of directory to maintain coherency for larger processor counts. In fact, as the number of cores on a single chip has grown, many have opted for a hybrid scheme even within a single chip, associating the directory information with the slices of a shared LLC. If the LLC is inclusive, the directory information can be associated with the tag information for each LLC block. The directory information can indicate individual L2 caches that have a copy of the shared data, or, if there is a shared structure connecting multiple L2s (like the ring bus structure in the Power processors), we need only designate the collection of processors that must be snooped. Alternatively, we can make LLC inclusive for all shared lines (but not other lines) or create a directory that indicates which L2 or L2-clusters need to be snooped. The AMD EPYC design creates a directory by duplicating the L2 tags.

Once we go beyond the bounds of a shared LLC, we need a single point to associate with the directory information for a given block. Since memory is also distributed in such designs, the easiest place to locate the directory is with the memory interface. Assuming that coherency is implemented at the LLC level, the directory need only keep track of the LLCs of those cores that may have a copy of a shared block. This means that the directory information per memory block is small (1−16 bits). It is also possible to keep this data in cached form, so it need not be kept for every memory block. Of course, going to a cache structure for the directory means that misses must be handled. Misses in a directory cache are most easily handled by assuming that the block is present and sending invalidates to every cluster.

While most designs use just a directory or a directory on top of snooping a subset of the cores, the roles can be reversed. For a small chip count, we might snoop at the directories kept at the LLC level. A hit in the LLC directory would then generate explicit invalidates to the L2 caches that have copies. The EPYC solution follows this approach, since it has at most two sockets and 2 LLCs.

Performance of a NUMA Multiprocessor on a WWW Search Application

As of 2022 all newer multicores are NUMA and many are NUCA with a distributed LLC, and of course, the nonuniformity gets larger in system with more cores. Because larger versions of such systems are used primarily in servers often providing interactive services, performance requires both high throughput and rapid response (usually with a quality-of-service requirement).

To examine performance and system tradeoffs, we focus on a workload based on the most time-consuming portion of interactive search (see Ayers, et al., [2018] for more details about the workload and the data we summarize here). This computation, which occurs primarily at the roots of a search tree, examines

individual segments (called shards) of the web index (which are built offline in batch mode). This portion of the search process is like several other portions, which are less time consuming. Compared to other workloads, such as the SPECint benchmarks, the search workload has a higher L2 instruction miss rate, while its overall CPI and LLC miss rates fall between the most memory-intensive benchmarks (e.g., mcf) and the more typical benchmarks in SPECint. The workload is highly parallel: its speedup on a 72-core Intel Xeon E7 system is linear when compared to an 8-core system. This speed-up is possible because the workload consists of request-level parallelism (see Chapter 6) and has minimal communication among processes (and hence little coherence traffic arising from shared data being read and written).

Data was collected on an Intel Xeon E788xx with the following characteristics:

Cores	18 per chip
Total cores (2 chips)	36
SMT Threads per core	2
Cache block size	64 B
L1I (per core)	32 KiB 4-way
L1D (per core)	32 KiB 8-way
L2 (per core)	256 KiB 8-way
Shared L3 (per socket)	45 MiB 16-way

Although there is little coherence traffic, this search application exercises the memory system heavily. Figure 5.18 shows the MPKI for L1, L2, and L3, as well as an estimate of the miss cost obtained by multiplying the average miss penalty by the MPKI. This overestimates the actual stall time since some of the miss time will be hidden by the processor. The importance of L3 misses means there is a strong correlation between CPI and the L3 miss rate, which we also observed in Chapter 2 when we looked at hardware prefetching. Recall that L3 misses, which access off-chip DRAM, are also 10 or more times the energy consumption of L3 hits.

Because of set associative caches, conflict misses are minor: eliminating them all by using a fully associative cache leads to only a 7% reduction in the miss rate in the two L1 caches and less than a 1% reduction in L2 and L3. For this long-running application (and hence small numbers of compulsory misses) with small numbers of coherence misses, it is the capacity misses that dominate. This observation is not surprising given the size of the data being used in a WWW search engine.

The use of a large L3 cache is one way to reduce capacity misses. If there is high spatial locality, increasing the block size could also decrease the capacity miss rate. Figure 5.19 shows the impact of varying the block size for each of the caches with this workload. Remember that because there is little sharing,

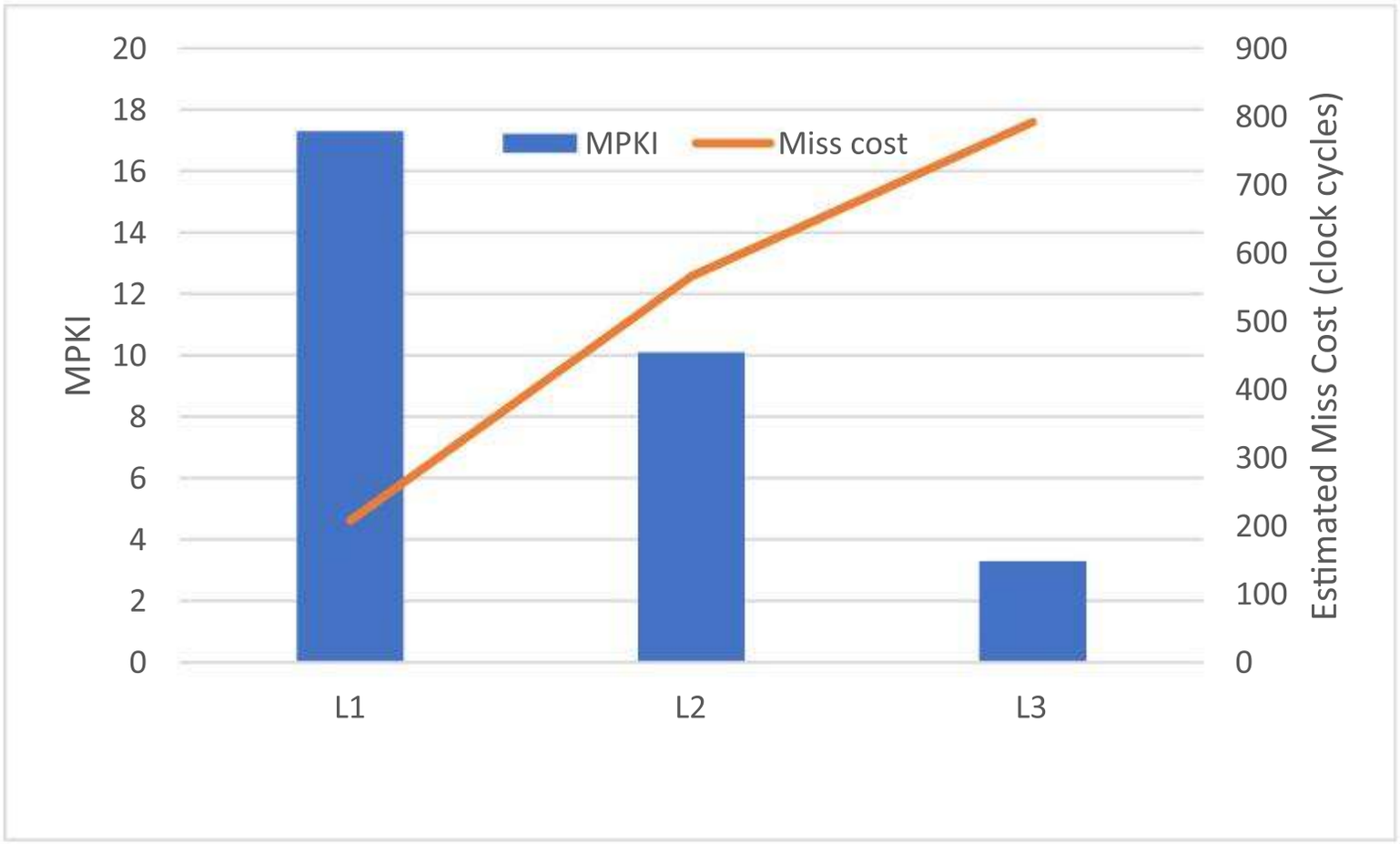


Figure 5.18 The columns show the MKPI for L1, L2, and L3, while the line shows the estimated miss cost on the right axis. Because the processor can more easily hide the L1 or L2 miss penalty (12 and 56 cycles, respectively) compared to the L3 miss penalty (240 cycles), the effective miss cost will be lower for L1 and L2 relative to that estimated miss cost for L3 shown in the chart. The estimate assumes that 2/3 of the misses to L3 and memory hit in the local L3 bank or local memory. No coherence transaction costs are included.

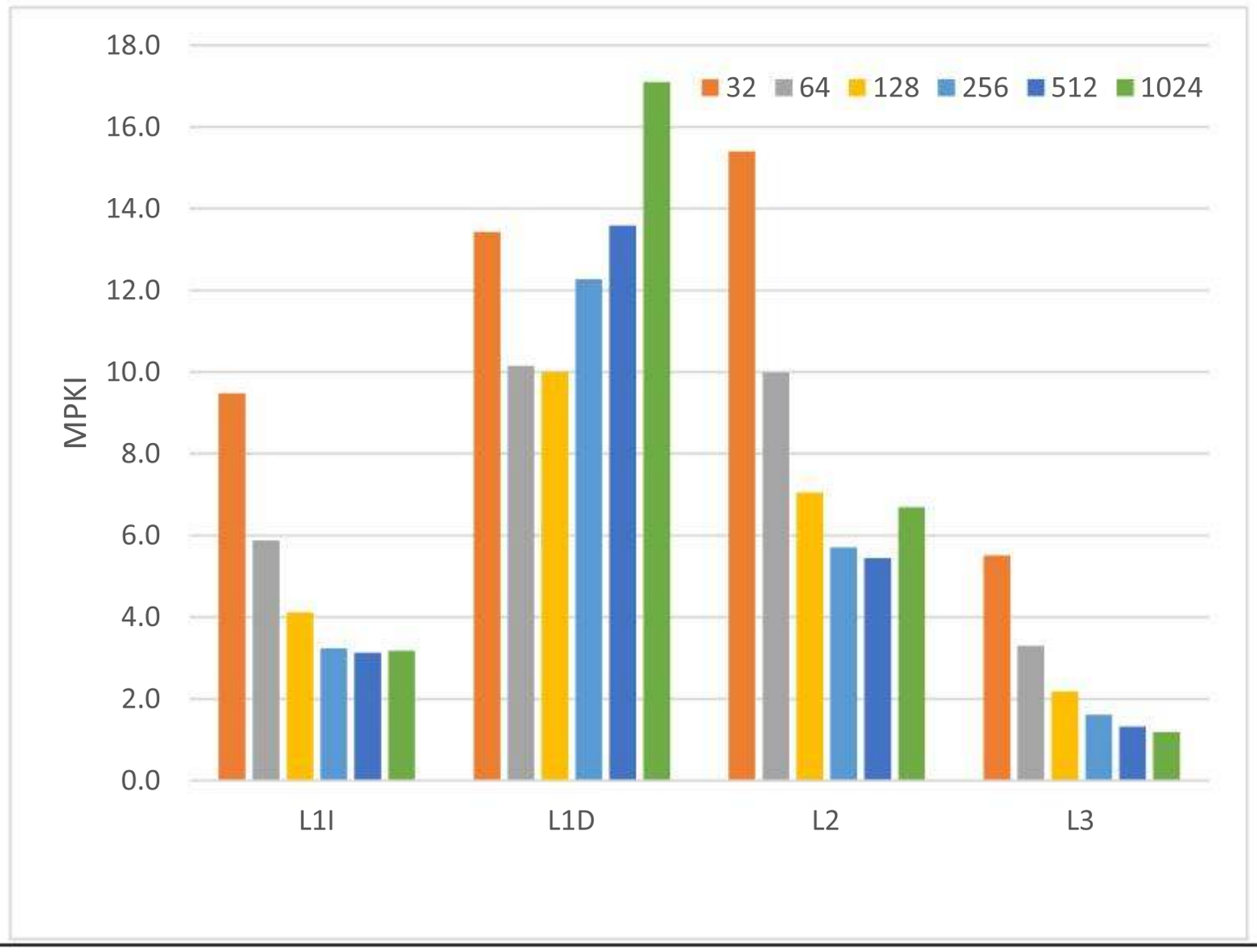


Figure 5.19 Miss rate for the various cache levels as block size is increased from 32 to 1024 bytes. The biggest reductions are seen when there is very high spatial locality (as in the L1 I cache) or when the cache is very large, as in L3.

the coherency misses, which would normally grow due to false sharing with larger blocks, are a minor effect. As we might suspect, increasing the block size has the biggest effect on misses for the L1 instruction cache, but it also shows significant gains in the large L3 cache. The L1 data cache shows the greatest sensitivity: initially reducing the miss rate by about 25% before increasing the miss rate at the larger block sizes.

Of course, while increasing the block size often decreases the miss rate, it increases the traffic to the next level of the hierarchy. Thus whether increasing the block size improves performance depends on the impact of potential contention from higher memory traffic, the higher cost of a miss (which in turn depends on whether the memory system returns the critical word first and allows the memory reference to precede), and the potential impact on power consumption, which is dominated by the number of misses, rather than the block size.

Figure 5.20 shows the memory traffic for larger block sizes relative to the traffic for a 32B block. Choosing the best block size, even considering just this one benchmark, is complex. For example, it appears that a larger block size for L1I makes sense, since the reduction in miss rate in going to 64B blocks is large and results in only a small increase in traffic. Because inclusion makes it tricky to have different block sizes in L1D and L2, the choice to increase the block size of L2 to 64 will result in a 1.5x increase in traffic to L2 and 1.3x increase from L2 to L3. Whether this is a good tradeoff depends on analyzing the complete system behavior on a wider set of benchmarks. Increasing the block size of L3, assuming inclusion is not required, seems more appealing. Making that change, however, needs to also consider how coherence information for the L2 caches is tracked. Nonetheless, the small increase in traffic (1.2x) and significant decrease in miss

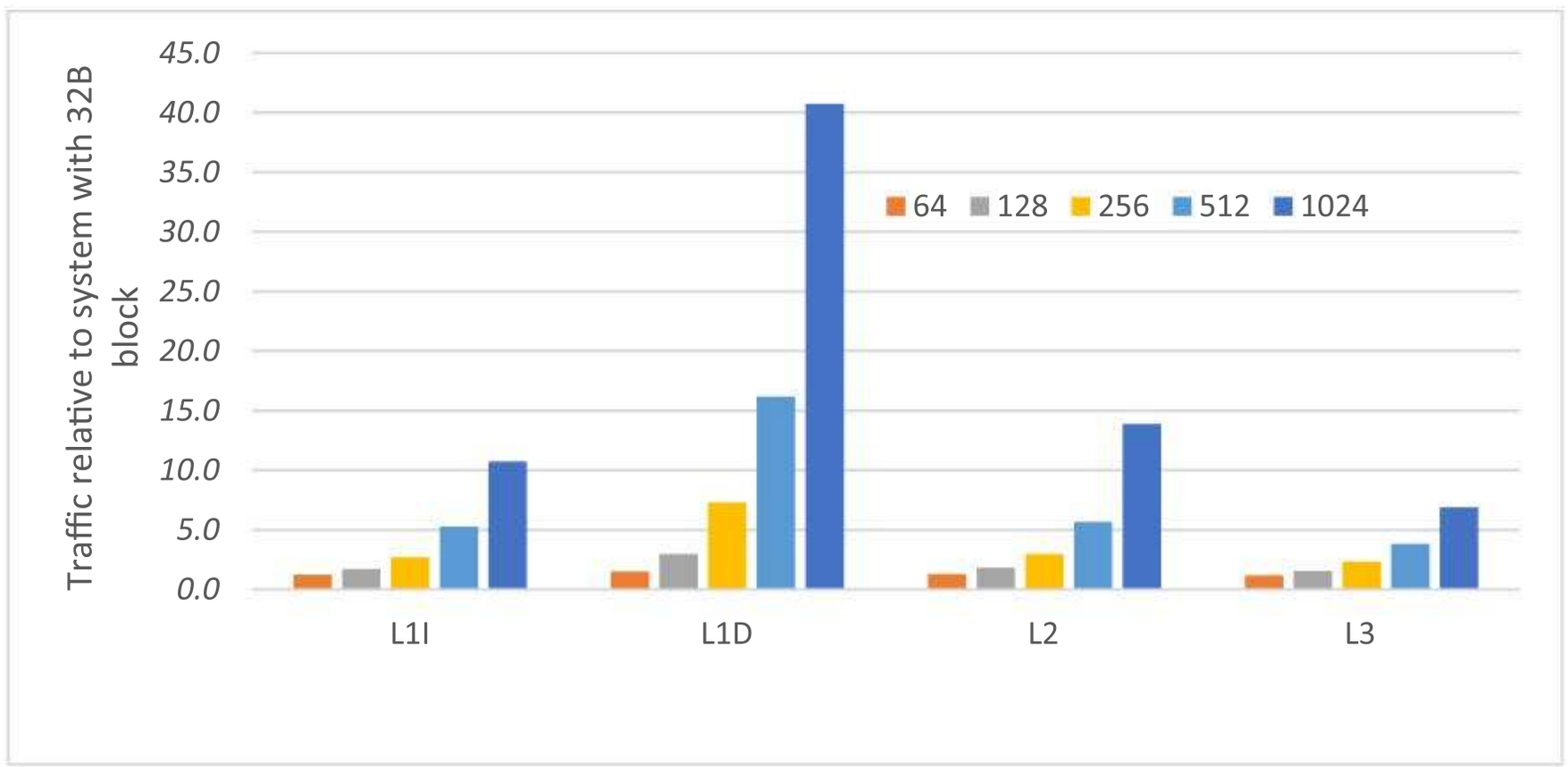


Figure 5.20 The relative traffic to the next level in the hierarchy as the block size is increased from 32B. The rate of traffic increase is determined by the decrease in miss rate as block size is increased. L1I and L3 show the biggest drops in miss rate and hence the smallest growth in traffic.

rate (1.7x) is compelling. For applications that have more intensive data sharing (see Appendix I for an analysis of the misses in scientific applications), increasing the block size is likely to generate more coherence misses, further reducing the advantages of a larger block size for L1D and L2.

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 a larger 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 several 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 whether 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 has 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 because 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 determine they have simultaneously set the variable.

There are a few 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 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 like 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 because it requires both a memory read and a write in a single, uninterruptible instruction. This requirement complicates the implementation of coherence because 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 though the instructions were atomic. The pair of instructions is effectively atomic if it appears as though 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. This is the approach used in the MIPS processors and in RISC V.

In RISC V the pair of instructions includes a special load called a *load reserved* (also called load linked or *load locked)* and a special store called a *store conditional*. Load reserved loads the contents of memory given by rs1 into rd and creates a reservation on that memory address. Store conditional stores the value in rs2 into the memory address given by rs1. If the reservation of the load is broken by a write to the same memory location, the store conditional fails and writes a nonzero to rd; if it succeeds, the store conditional writes 0. If the processor does a context switch between the two instructions, then the store conditional always fails.

These instructions are used in sequence, and because the load reserved returns the initial value and the store conditional returns 0 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of x1 with the value in x4:

```
try: mov x3,x4 ;mov exchange value
lr x2,0(x1) ;load reserved from x1
sc x3,0(x1),x5 ;stores x3, x5 = 0 if success
bnez x5,try ;branch store fails—try again
mov x4,x2 ;put load value in x4
```

At the end of this sequence, the contents of `x4` and the memory location specified by `x1` have been atomically exchanged. Anytime a processor intervenes and modifies the value in memory between the `lr` and `sc` instructions, the `sc` returns a nonzero in `x5`, causing the code sequence to try again.

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

```
try: lr x2,0(x1) ;load reserved 0(x1)
addi x3,x2,1 ;increment
sc x3,0(x1),x4 ;store conditional
bnez x4,try ;branch if store fails
```

These instructions are typically implemented by keeping track of the address specified in the `lr` instruction in a register, often called the *reserved register*. If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (e.g., by another `sc`), the link register is cleared. The `sc` instruction simply checks that its address matches that in the reserved register. If so, the `sc` succeeds; otherwise, it fails. Because the store conditional will fail after either another attempted store to the load reserved address or any exception, care must be taken in choosing what instructions are inserted between the two instructions. 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 reserved 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, 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 `x1`. It uses `EXCH` as a macro for the atomic exchange sequence from page 414:

```
lockit: addi x2,R0,#1 ; lock locked value
EXCH x2,0(x1) ;atomic exchange
bnez x2,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 soon. 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 preceding loop 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 because 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 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 which can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter). The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again. Here is the code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit: ld x2,0(x1) ;load of lock
bnez x2,lockit ;not available-spin
addi x2,R0,#1 ;load locked value
EXCH x2,0(x1) ;swap
bnez x2,lockit ;branch if lock wasn't 0
```

Let's examine how this "spin lock" scheme uses the cache coherence mechanisms. Figure 5.21 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.

This example shows another advantage of the load reserved/store conditional primitives: the read and write operations are explicitly separated. The load reserved 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 (`x1` has the address of the lock, the `lr` has replaced the `LD`, and the `sc` has replaced the `EXCH`):

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 5.21 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—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 because 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.

```
lockit: lr x2,0(x1) ;load reserved
bnez x2,lockit ;not available-spin
addi x2,R0,#1 ;locked value
sc x2,0(x1),x3 ;store
bnez x3,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 a processor must see a value that has been updated by another processor. Because 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? Because the only way to "observe the writes of another processor" is through reads, the question becomes, what properties must be enforced among reads and writes to different locations by different processors?

Although the question of how consistent memory must be seems simple, it is remarkably complicated, as we can see with a simple example. Here are two code segments from processes `P1` and `P2`, shown side by side:

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

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

The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as though 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 a 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 to different memory locations. In our example we must delay the read of A or the read of 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 many processors or long interconnect delays, as we can see in the following example.

Example Suppose we have a processor where a write miss takes 60 cycles to establish ownership, 40 cycles to issue the invalidates after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued. If 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. Therefore the total time for the write is $60 + 40 + 80 = 180$ cycles. In comparison, the ownership time is only 60 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 in which 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 a multiprocessor. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the multiprocessor and the type of synchronization primitive.

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 though the hardware implemented sequential consistency.

Relaxed Consistency Models: The Basics and Release Consistency

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 though the processor were sequentially consistent. There are a variety of relaxed models that are classified according to what read and write orderings they relax. We specify the orderings by a set of rules of the form X → Y, meaning that operation X must complete before operation Y is done. Sequential consistency requires maintaining all four possible orderings:

$$R \rightarrow W, R \rightarrow R, W \rightarrow R, \text{and } W \rightarrow W.$$

The relaxed models are defined by the subset of four orderings they relax: Relaxing only the W → R ordering yields a model known as *total store ordering* or *processor consistency*. Because this model retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.

Relaxing both the W → R ordering and the W → W ordering yields a model known as *partial store order*.

Relaxing all four orderings yields a variety of models including *weak ordering*, the PowerPC consistency model, and *release consistency*, the RISC V consistency model.

By relaxing these orderings, the processor may obtain significant performance advantages, which is the reason that RISC V, ARMv8, and the C++ and C language standards chose release consistency as the model.

Release consistency distinguishes between synchronization operations that are used to *acquire* access to a shared variable (denoted S_A) and those that *release* an object to allow another processor to acquire access (denoted S_R). Release consistency is based on the observation that in synchronized programs, an acquire operation must precede a use of shared data, and a release operation must follow any updates to shared data and precede the time of the next acquire. This property allows us to slightly relax the ordering by observing that a read or write that precedes an acquire need not complete before the acquire, and that a read or write that follows a release need not wait for the release. Thus the orderings that are preserved involve only S_A and S_R, as shown in Figure 5.22; as the example in Figure 5.23 shows, this model imposes the fewest orders of the five models.

Release consistency provides one of the least restrictive models that is easily checkable and ensures that synchronized programs will see a sequentially consistent execution. Although most synchronization operations are either an acquire or a release (an acquire normally reads a synchronization variable and atomically updates it, and a release usually just writes it), some operations, such as a barrier, act as both an acquire and a release and cause the ordering to be equivalent to weak ordering. Although synchronization operations always ensure that previous writes have completed, we may want to guarantee that writes are completed without an identified synchronization operation. In such cases an explicit instruction, called FENCE in RISC V, is used to ensure that all previous instructions in that thread have completed, including completion of all writes to memory and associated invalidates. As we mentioned earlier, tracking these operations in a

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	R→R, R→W, W→R, W→W	S→W, S→R, R→S, W→S, S→S
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	R→R, R→W, W→W	S→W, S→R, R→S, W→S, S→S
Partial store order	SPARC	R→R, R→W	S→W, S→R, R→S, W→S, S→S
Weak ordering	PowerPC		S→W, S→R, R→S, W→S, S→S
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		S_A→W, S_A→R, R→S_R, W→S_R, S_A→S_A, S_A→S_R, S_R→S_A, S_R→S_R

Figure 5.22 The orderings imposed by various consistency models are shown for both ordinary accesses and synchronization accesses. The models grow from most restrictive (sequential consistency) to least restrictive (release consistency), allowing increased flexibility in the implementation. The weaker models rely on fences created by synchronization operations, as opposed to an implicit fence at every memory operation. S_A and S_R stand for acquire and release operations, respectively, and are needed to define release consistency. If we were to use the notation S_A and S_R for each S consistently, each ordering with one S would become two orderings (e.g., S → W becomes S_A → W, S_R → W), and each S → S would become the four orderings shown in the last line of the bottom-right table entry.

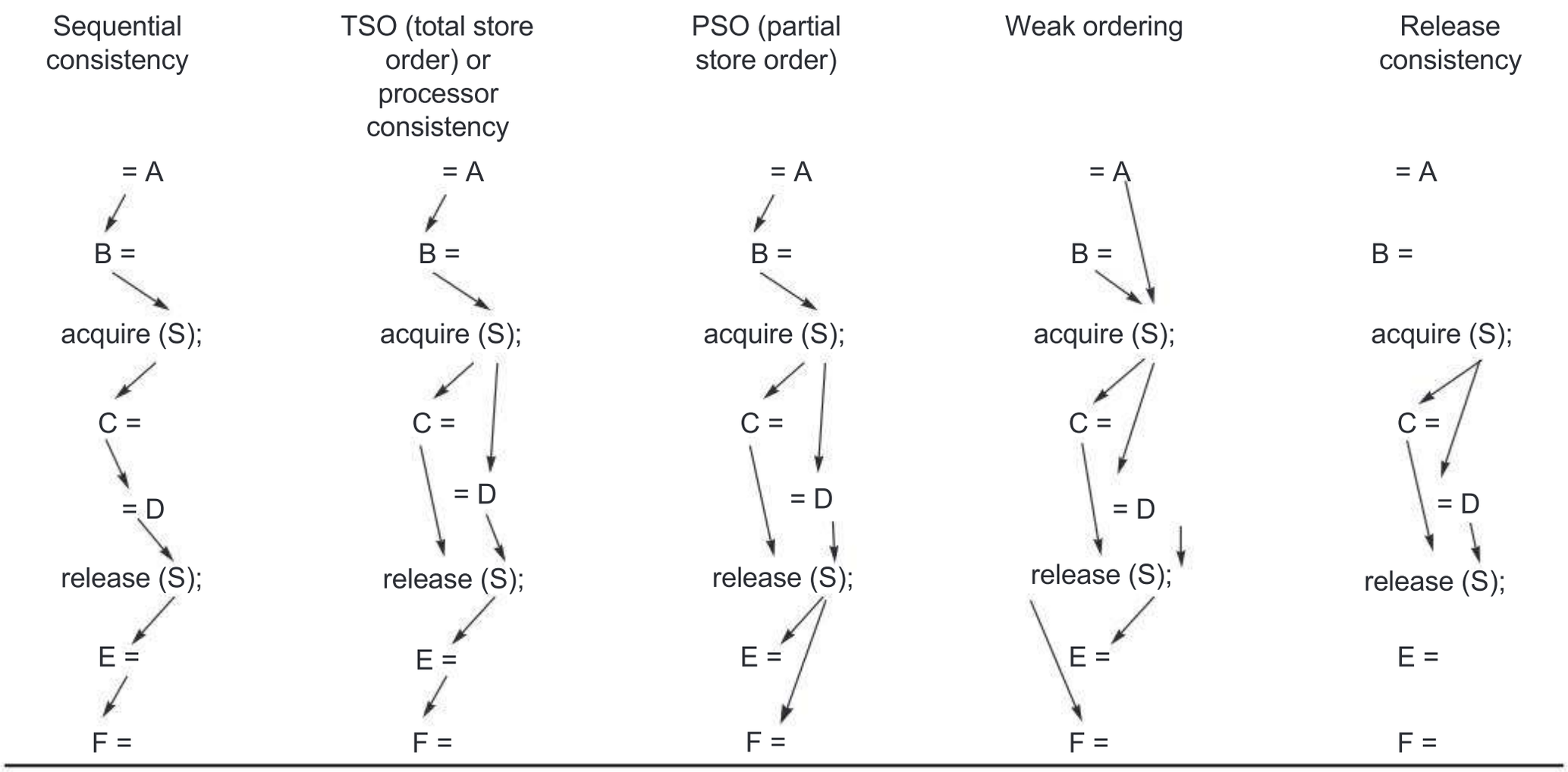


Figure 5.23 These examples of the five consistency models discussed in this section show the reduction in the number of orders imposed as the models become more relaxed. Only the minimum orders are shown with arrows. Orders implied by transitivity, such as the write of C before the release of S in the sequential consistency model or the acquire before the release in weak ordering or release consistency, are not shown.

larger multiprocessor without a broadcast medium requires that we track invalidations so that we know when they are completed. 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).

5.7 Cross-Cutting Issues

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

Compiler Optimization and the Consistency Model

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

One open question is how successful compiler technology will be in optimizing memory references to shared variables. The state of optimization technology and the fact that shared data are often accessed via pointers or array indexing have limited the use of such optimizations. If this technology were to become available and lead to significant performance advantages, compiler writers would want to be able to take advantage of a more relaxed programming model. This possibility and the desire to keep the future as flexible as possible led the RISC V designers to opt for release consistency, after a long series of debates.

Using Speculation to Hide Latency in Strict Consistency Models

As we saw in Chapter 3, speculation can be used to hide memory latency. It can also be used to hide latency arising from a strict consistency model, giving much of the benefit of a relaxed memory model. The key idea is for the processor to use

dynamic scheduling to reorder memory references, letting them possibly execute out of order. Executing the memory references out of order may generate violations of sequential consistency, which might affect the execution of the program. This possibility is avoided by using the delayed commit feature of a speculative processor. Assume the coherency protocol is based on invalidation. If the processor receives an invalidation for a memory reference before the memory reference is committed, the processor uses speculation recovery to back out of the computation and restart with the memory reference whose address was invalidated. Of course, the processor must also ensure that such an approach does not create new opportunities for using speculation to do a side-channel attack (see Chapter 3).

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

Hill (1998) advocated the combination of sequential or processor consistency together with speculative execution as the consistency model of choice. His argument has three parts. First, an aggressive implementation of either sequential consistency or processor consistency will gain most of the advantage of a more relaxed model. Second, such an implementation adds very little to the implementation cost of a speculative processor. Third, such an approach allows the programmer to reason using the simpler programming models of either sequential or processor consistency. The MIPS R10000 design team had this insight in the mid-1990s and used the R10000's out-of-order capability to support this type of aggressive implementation of sequential consistency.

Inclusion and Its Implementation

All multiprocessors use multilevel cache hierarchies to reduce both the demand on the global interconnect and the latency of cache misses. If the cache also provides *multilevel inclusion*—every level of cache hierarchy is a subset of the level farther away from the processor—then we can use the multilevel structure to reduce the contention between coherence traffic and processor traffic that occurs when snoops and processor cache accesses must contend for the cache. Many multiprocessors with multilevel caches enforce the inclusion property, although several recent multiprocessors have chosen not to enforce inclusion.

At first glance, preserving the multilevel inclusion property seems trivial. Consider a two-level example:

Any miss in L1 either hits in L2 or generates a miss in L2, causing it to be brought into both L1 and L2. Likewise, any invalidate that hits in L2 must be sent to L1, where it will cause the block to be invalidated if it exists.

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

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

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

To maintain inclusion with multiple block sizes, we must probe the higher levels of the hierarchy when a replacement is done at the lower level to ensure that any words replaced in the lower level are invalidated in the higher-level caches; different levels of associativity create the same sort of problems. Baer and Wang (1988) described the advantages and challenges of inclusion in detail, and in 2017 most designers have opted to implement inclusion, often by settling on one block size for all levels in the cache. Recent Intel processors (including the i9 and high-end Xeons) and the AMD EPYC do not enforce inclusion for L3 (but do for L1 and L2). This decision allows L2 and L3 to hold different data increasing the effectiveness of L3, which on a per-core basis is only slightly larger than L2.

Multiprocessor Performance Gains from Multithreading

In this section we briefly look at a study of the effectiveness of using multithreading on two multicore processors: an Intel Xeon E7 8800 and a Power8 system. We use the highly parallel web search benchmark we discussed at the end of Section 5.4. The benefit to be gained from SMT depends on the processor count, the number of threads supported, and the ILP pipeline. For these two processors, here are the most relevant characteristics:

Characteristic	Intel Xeon E7	IBM Power8
Cores	18	12
SMT threads per core	2	8
Issues per clock	4	8
Functional units	8	16

Recall that the web search application has lots of request-level parallelism and little interprocess communication, so it gets linear speed-up on a Xeon E7 scaling from 8 to 72 cores. Figure 5.24 shows the speed-up with an increase in the thread count relative to the single-threaded version. The Power8 with its 8-thread support gets a larger overall benefit, although the benefit may be closely tied to the higher issue rate and larger number of functional units.

5.8 Putting It All Together: Multicore Processors and Their Performance

For roughly 15 years, multicore has been the primary focus for scaling performance, although the implementations vary widely, as does their support for larger

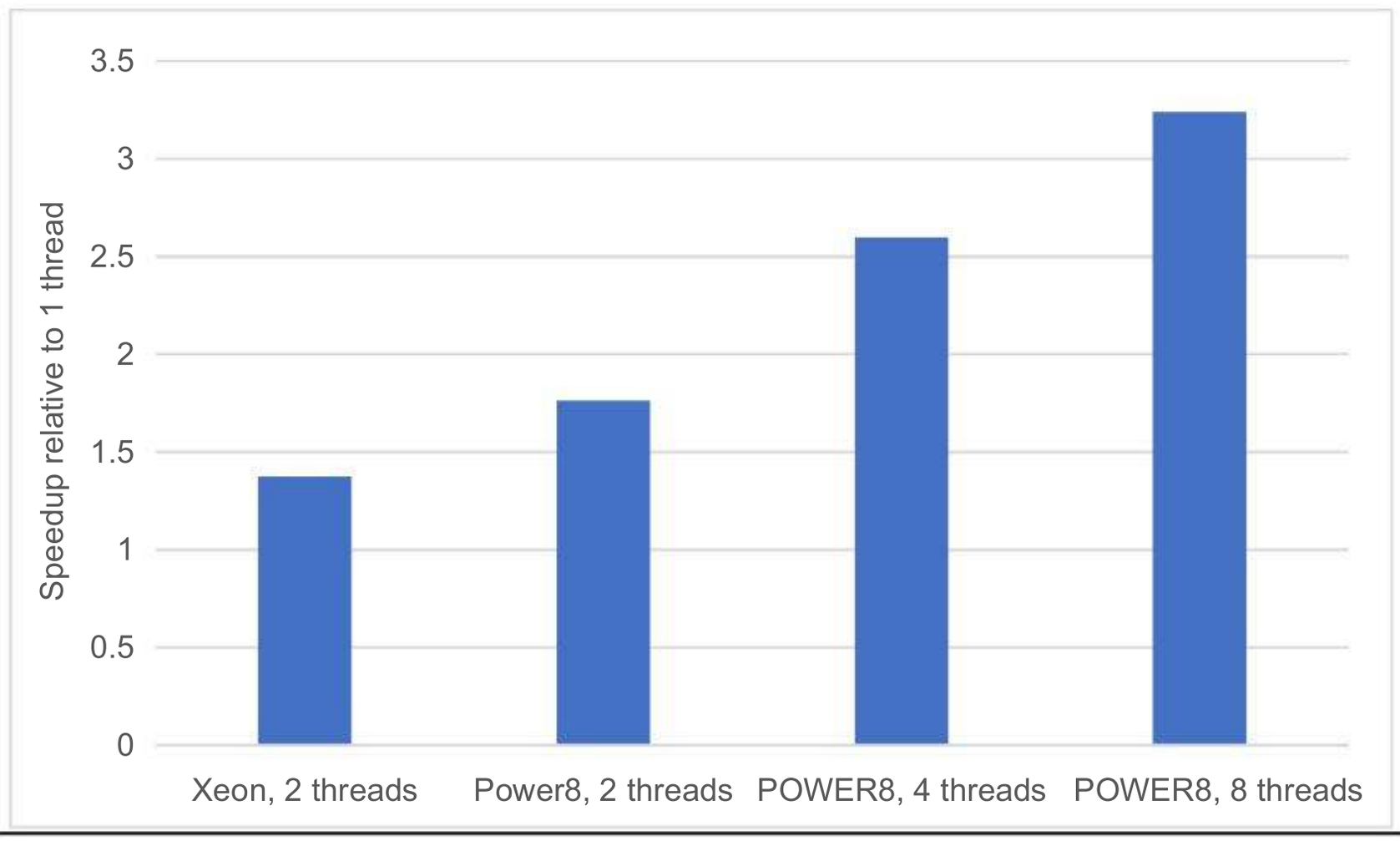


Figure 5.24 Speed-up from SMT relative to single thread running on the same number of cores (18 for Xeon and 12 for Power8). The Power8 is the only processor to offer more than 2 threads in an SMT core. In this highly parallel application with the wide issue, large functional unit count of the Power8, the performance advantages are significant.

multichip multiprocessors. In this section we examine the design of three different multicores, the support they provide for larger multiprocessors, and the scalability of some of these systems in terms of both performance and energy efficiency.

Three Multicore Server Processors

Figure 5.25 shows the key characteristics of three multicore processors designed for server applications and available in systems from 2019 through 2022. The Intel

Feature	IBM Power10	Intel Xeon Platinum	AMD EPYC
Cores/chip;MCM	4–15 per chip; 8–30 per socket	4–60 per chip; 8–120 per socket	8 per chip; 64 per socket
Multithreading	SMT	SMT	SMT
Threads/core	8	2	2
Clock rate	4.15 GHz	2.2–3.6 GHz	3.5 GHz
L1 I cache	96 KiB per core	32KiB per core	32 KiB per core
L1 D cache	64 KiB per core	32 KiB per core	32 KiB per core
L2 cache	2 MiB per core	1 MiB per core	512KiB per core
L3 cache	L3: 8 MiB per core; shared with nonuniform access time	22–77 MiB @ 1.375 MiB per core; shared, with larger core counts	4MiB per core to 12 MiB per core using stacked memory; shared
Inclusion	L2 inclusive	L2 inclusive	L3 for shared blocks
Multicore coherence protocol	Extended MESI with behavioral and locality hints (13-states)	MESI F: an extended form of MESI allowing direct transfers of clean blocks	MDOEFSI: an extended form of MOESI with dirty and forward states
Multichip coherence implementation	Hybrid strategy primarily snooping with limited directory	Hybrid strategy with snooping and directory at L3	Hybrid strategy with snooping and directory (copy of L2 tags) at L3
Multiprocessor interconnect support	Can connect up to 16 processor chips with 1 or 2 hops to reach any processor	Up to 8 processor chips directly via UPI	Infinity Fabric connects 4 chiplets per socket and up to 2 sockets.
Processor chip range	1–16	2–8	1–2
Core count range	4–240	8–480	8–128

Figure 5.25 Summary of the characteristics of three recent high-end multicore processors (2020–2023 releases) designed for servers. The table shows the range of processor counts, clock rates, and cache sizes within each processor family. All these processors have a NUCA as well as a NUMA structure. The L3 access time is given for the local slice and a remote slice. The memory access time shows an access time to local memory and remote memory. All access times are average. The last row shows the range of configured systems with published performance data (such as SPECintRate) with both processor chip counts and total core counts. Note that while Intel Xeons are available with 56 cores, the processor using the most recent microarchitectures tops out at 40 cores. Most of these systems are available in a chiplet form, with the IBM and Intel designs also supporting larger multiprocessors including multiple chiplets.

Xeon Platinum uses a core similar to the Skylake microarchitecture but has a slightly slower clock rate than the desktop processors (power is the limitation), includes a larger L3 cache, and many more cores. The Power10 is the newest in the IBM Power series and features more cores and bigger caches. The AMD EPYC Milan series has 8 cores per chiplet and 4 chiplets per socket in this example. Because these processors are configured for multicore and multiprocessor servers, they are available as a family, varying processor count, cache size, and so on, as the figure shows.

These three systems show a range of techniques both for connecting the on-chip cores and for connecting multiple processor chips or chiplets.

Figure 5.26 shows how the Power10 and Xeon Platinum chips are organized. Each core in the Power10 has an 8 MiB bank of L3 directly connected; other banks are accessed via the interconnection network, which has 8 separate buses.

Part B of Figure 5.26 shows how the Xeon Platinum processor chip is organized when there are larger core counts (24 cores are shown in this figure). Two sets of rings connect a group of cores, each with its section of L3 and the coherency logic for a portion of the address space. Any cache bank or core is accessible from any other core by traversing one or two rings. The Xeon Platinum provides 3 UPI Interconnect links for connecting multiple Xeons.

The single-chip version of the AMD Eypc Milan is the simplest since each chiplet has only eight cores. The eight cores and an 8-bank L3 are connected by a bidirectional ring that is 32 bytes wide. The connection of other chiplets in the same socket is provided with the Infinity Path which fully connects the eight multicore chips in a socket. All three multicores are NUCAs, since the time to access L3 will depend on what bank is accessed.

Multiprocessors consisting of these multicore chips use a variety of different interconnection strategies, as Figure 5.27 shows. The Power10 design provides support for connecting 16 Power10 chips for a total of 240 cores. The intragroup connections create a completely connected module of 4 processor chips, and the intergroup links are used to connect each processor chip to the 3 chips in each of the other modules. The UPI is used to interconnect multiple Xeon chips. In an 8-socket multiprocessor, which with the latest announced Xeon could have 448 cores, the 3 UPI links on each processor are connected to 3 neighbors. Figure 5.27 shows how 8 Xeon processors can be connected; like the Power10, this leads to a situation where every processor is one or two hops from every other processor. A few companies have built versions of Xeon-based multiprocessors that interface to the UPI network and extend the network and coherence mechanisms to allow larger processor counts. The EPYC design connects to 8 chiplets in a socket with the Infinity Path. As Figure 5.27 shows, the chiplets in one socket are fully connected. Two sockets can be connected using the unused Infinity Path connections, resulting in a topology where every chiplet is one or two hops away.

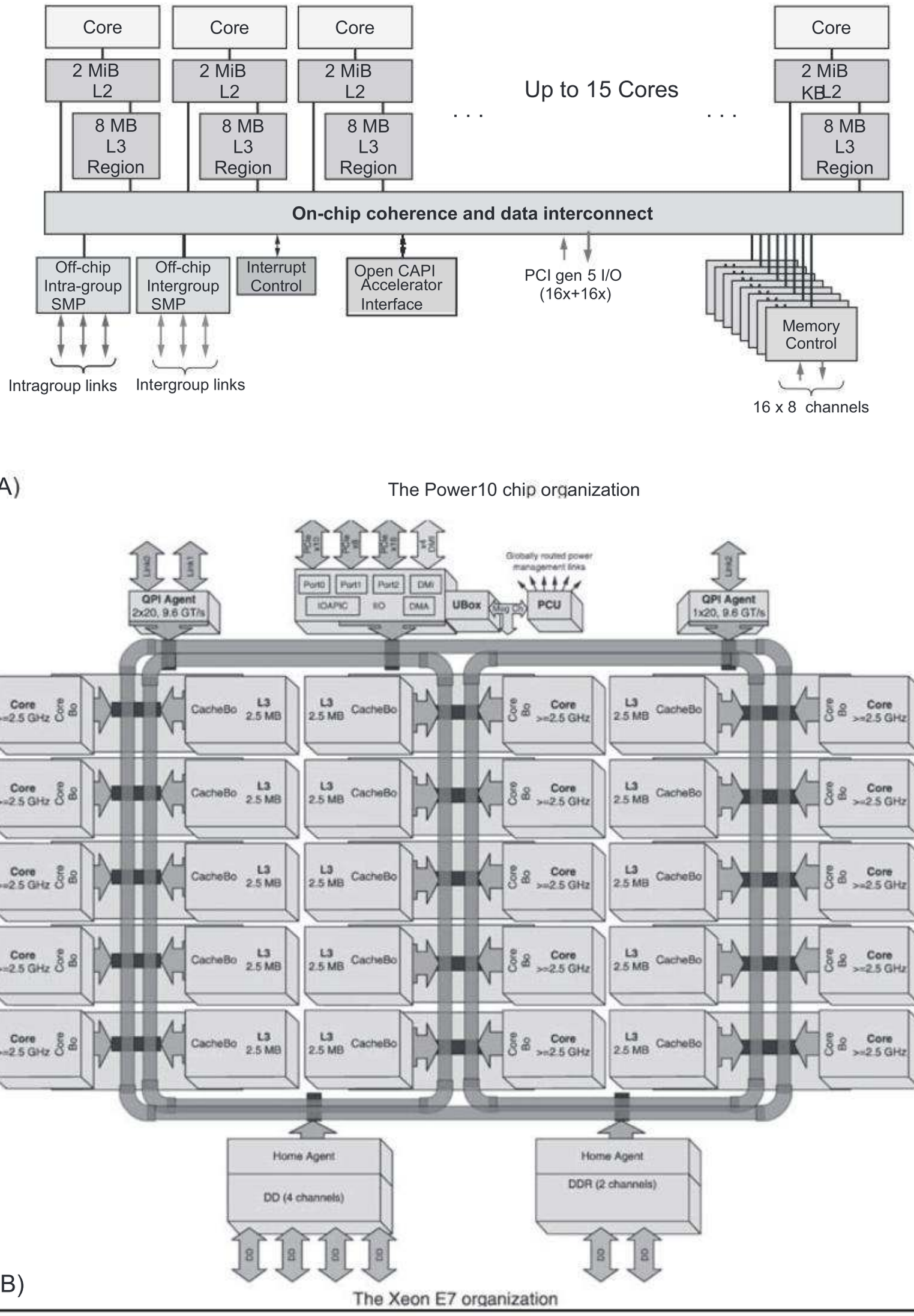
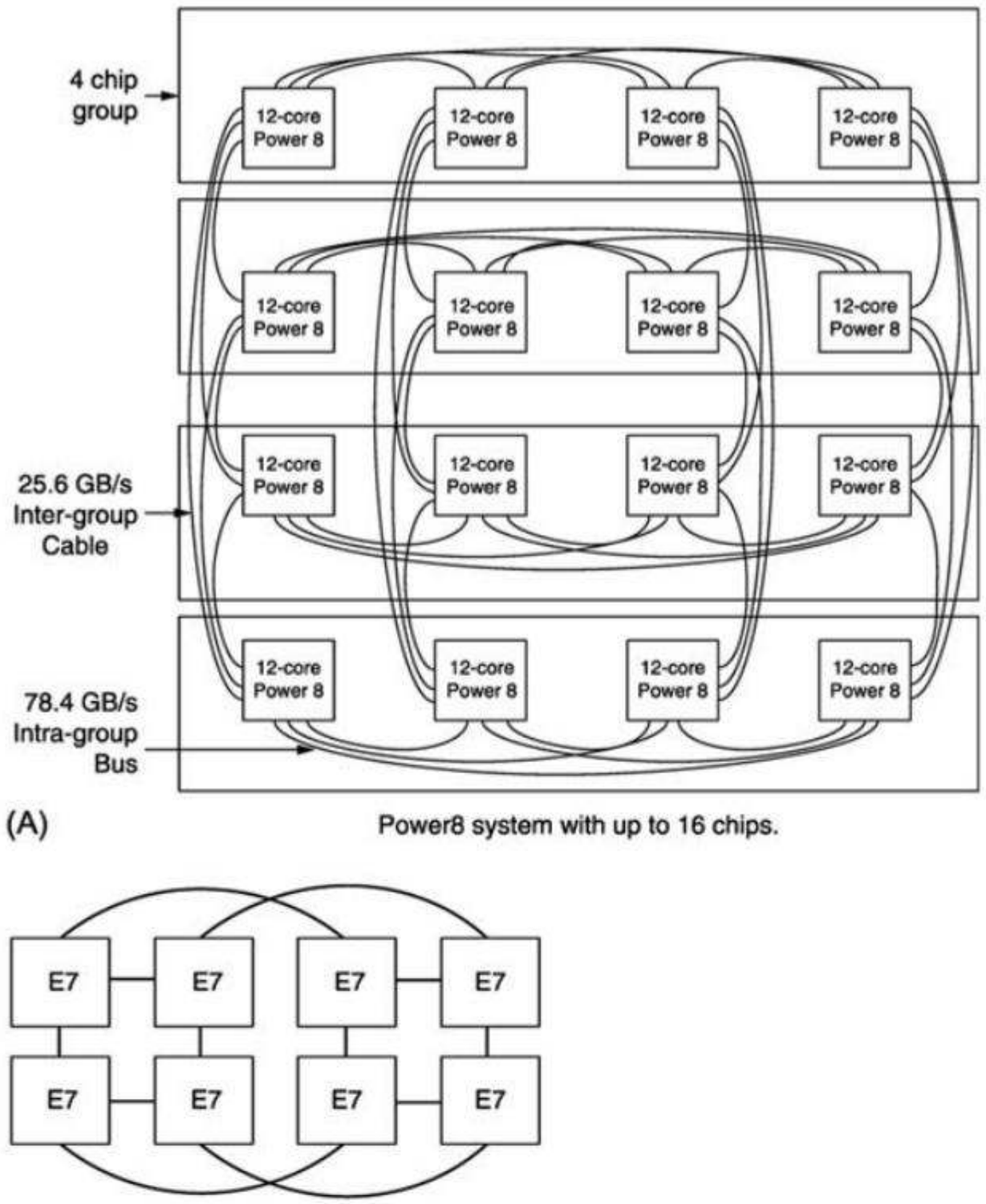
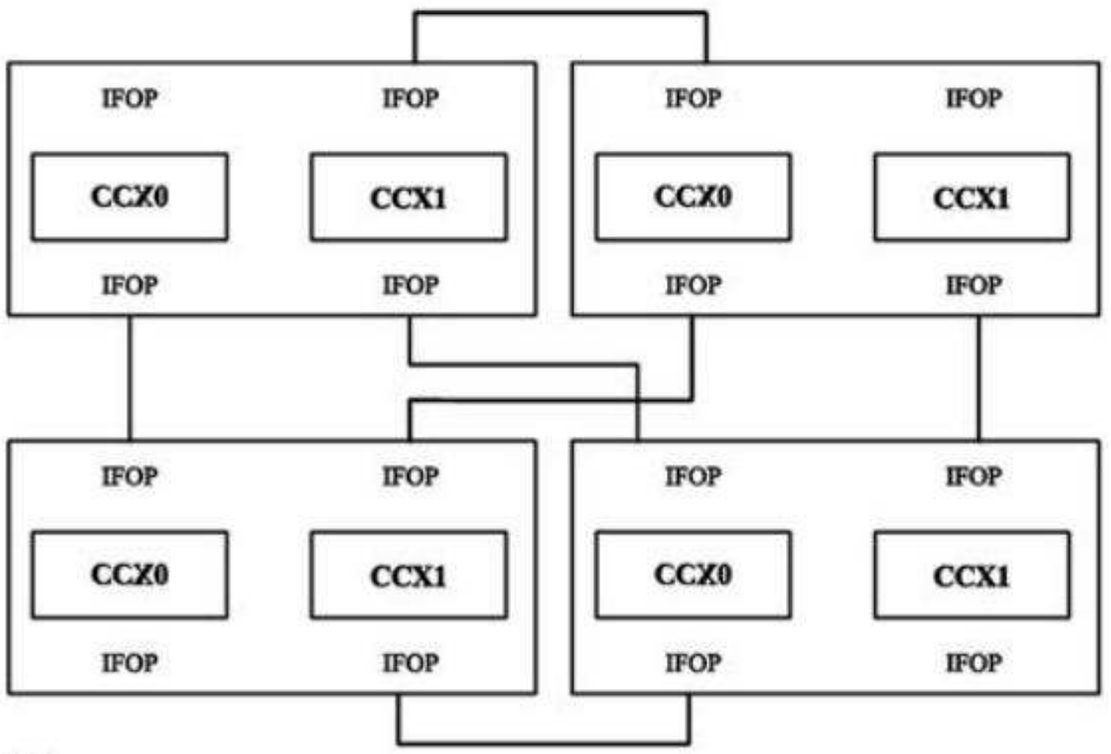


Figure 5.26 The on-chip organization of the IBM Power 10 and Intel Xeon E7 are shown.



(A) Power8 system with up to 16 chips.

(B) Xeon E7 system showing with up to 8 chips.



(C)

Figure 5.27 The system architecture for three multiprocessors (Power10, Xeon Platinum, and AMD EPYC) built from multicore chips. All three designs fully connect four chips, which are reachable in one hop and require another hop to get to a different group of four, so every multicore is one or two hops from every other. Thus connecting to the cache of a remote multicore (for a retrieval or invalidation) will take one or two hops through the intercore fabric and then a trip through the intracore fabric (and a similar return path if data is being retrieved). This diagram shows the fully connected four-chip, 32-core AMD EPYC (each CCX is 4 cores), which serves as the building block for a system with 64 cores in a socket and up to two sockets. The Infinity Path (IFOP) links fully connect four multicores. The unused IFOP links can each connect another 32-core module.

Performance of Multicore-Based Multiprocessors on a Multiprogrammed Workload

First, we compare the performance scalability of our three multicore processors using SPECintRate, up to the largest single system configuration submitted to SPEC (128–220 cores). Figure 5.28 shows how the performance scales relative to the performance of the smallest configuration of 15 or 16 cores, which we assume has perfect speed-up of 15 or 16. This figure does *not* show performance among these different processors. For example, the 16-core AMD EPYC performance is 1.14 faster than the Intel Xeon performance. Figure 5.28 only shows how the performance scales for each processor family as additional cores are added.

Both the AMD and Intel systems show reduced speed-up as the core counts increase. For example, at 240 cores, the Xenon system has a speedup relative to

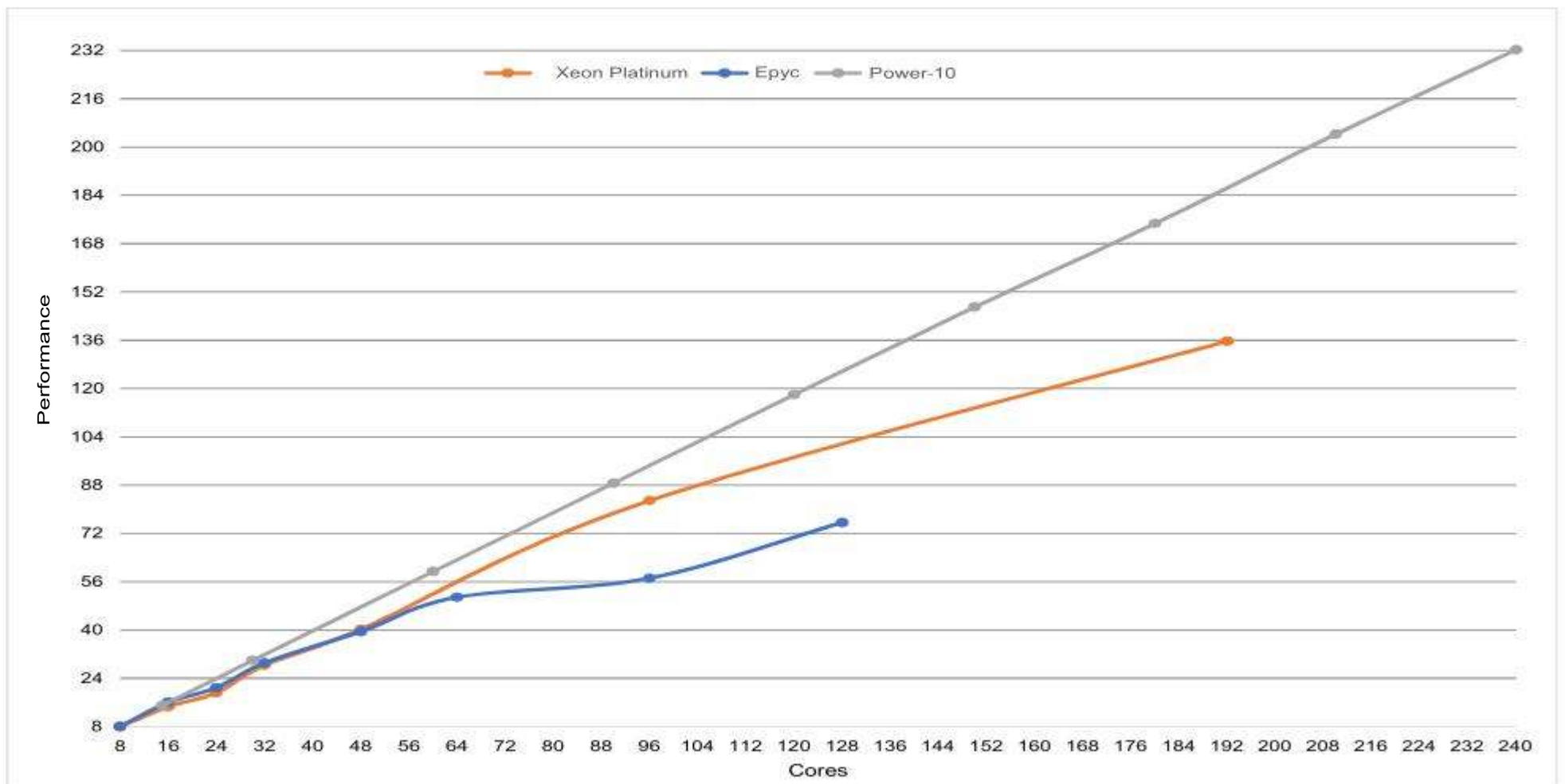


Figure 5.28 The performance scaling on the SPECintRate benchmarks for the Intel Xeon Platinum, AMD EPYC, and IBM Power10 for various core counts. Performance for each processor is plotted relative either to the 16-core configuration (for Intel and AMD) or 15-core version (for IBM) and assuming the 16- or 15-core configuration had perfect speedup. Although this chart shows how a given multiprocessor scales with additional cores, it does not supply any data about performance among processors. Note that at larger core counts for AMD and Intel, the effective speedup is about 50% to 60% of ideal, even when assuming perfect speed-up on 16-core processor. The IBM design shows much better scaling: at 220 cores it achieves 97% efficiency relative to the 15-core version. Due to a better scaling and a slightly better performance per core, a 210-core Power10 has 1.3x the throughput of a 240-core Xeon Platinum or about 50% more performance per core; our estimate is that 70%–80% of this advantage comes from better core performance and 20%–30% come from better scaling to larger core counts. Due to availability of released data and variations in core configurations, the Intel and AMD measurements may use results from slightly different processor configurations.

the 16-core system of 157.8, rather than 240. This represents about 70% efficiency compared to the 8-core results (which of course are unlikely to show perfect speed-up versus a single core). The AMD EPYC system is faster with small numbers of cores, but at 64 cores, it encounters performance limitations. These limits could arise due to systems issues as well as architectural limitations. At 128 cores, the EPYC system shows about 66% efficiency, compared to the 16-core version. By this metric of speed-up relative to the smallest configuration, the IBM Power10 design has the best scalability, with 97% efficiency at 220 cores. Since SPECintRate is not significantly influenced by cache coherency issues, memory and interconnect bandwidth probably play a significant role.

Scalability of a Xeon Platinum on Two Different Workloads

In this section we briefly examine the scalability of the Xeon Platinum multiprocessors on a scientific parallel processing workload and on a server workload based on a Java server, which measures performance as well as power.

SPEComp2012 is a collection of 14 scientific and engineering programs written with the OpenMP standard for shared-memory parallel processing. The codes are written in Fortran, C, and C++ and range from fluid dynamics to molecular modeling to image manipulation. Unlike the SPECintRate benchmark, the processes regularly communicate through shared address space. Like the previous results, Figure 5.29 shows performance assuming linear speedup on the smallest configuration (28 cores). For the smaller configurations up to 72 cores, the system gets linear speed-up relative to the 28-core system. A reduction to about 90% efficiency is seen in the range of 92 to 108 cores before a drop in efficiency to about 50% at 172—220 cores.

Our last comparison looks at the performance and power efficiency of Xeon systems using the SPECpower2008 benchmark. SPECpower models a server running multiple streams of Java. It exercises Java VMs, garbage collection, and Just-in-Time (JIT) compilers. The workload scales with system size, modeling a server in real use. Performance and power are reported for workloads ranging from 10% to 100%. The latter is the maximum workload the system can handle while maintaining quality of service guarantee and is scaled to the capacity of the computer (larger configurations run a larger workload). Figure 5.30 shows the increase in throughput and the energy efficiency (performance/average power) for the maximum workload from 28 to 224 cores. For this highly parallel, scaled benchmark, the Xenon Platinum shows essentially perfect speed-up from 28 to 224 cores. At first glance at the chart, you might think energy efficiency drops precipitously at the 224-core configuration; look carefully at the chart and check the caption.

SPECpower also measures the energy efficiency by measuring how the average power consumption at loads from Idle to 100%. Figure 5.31 shows this measure for a 224-core configuration of Xenon Platinum. Notice that at 50% of the load the system has an average power consumption that is 60% of

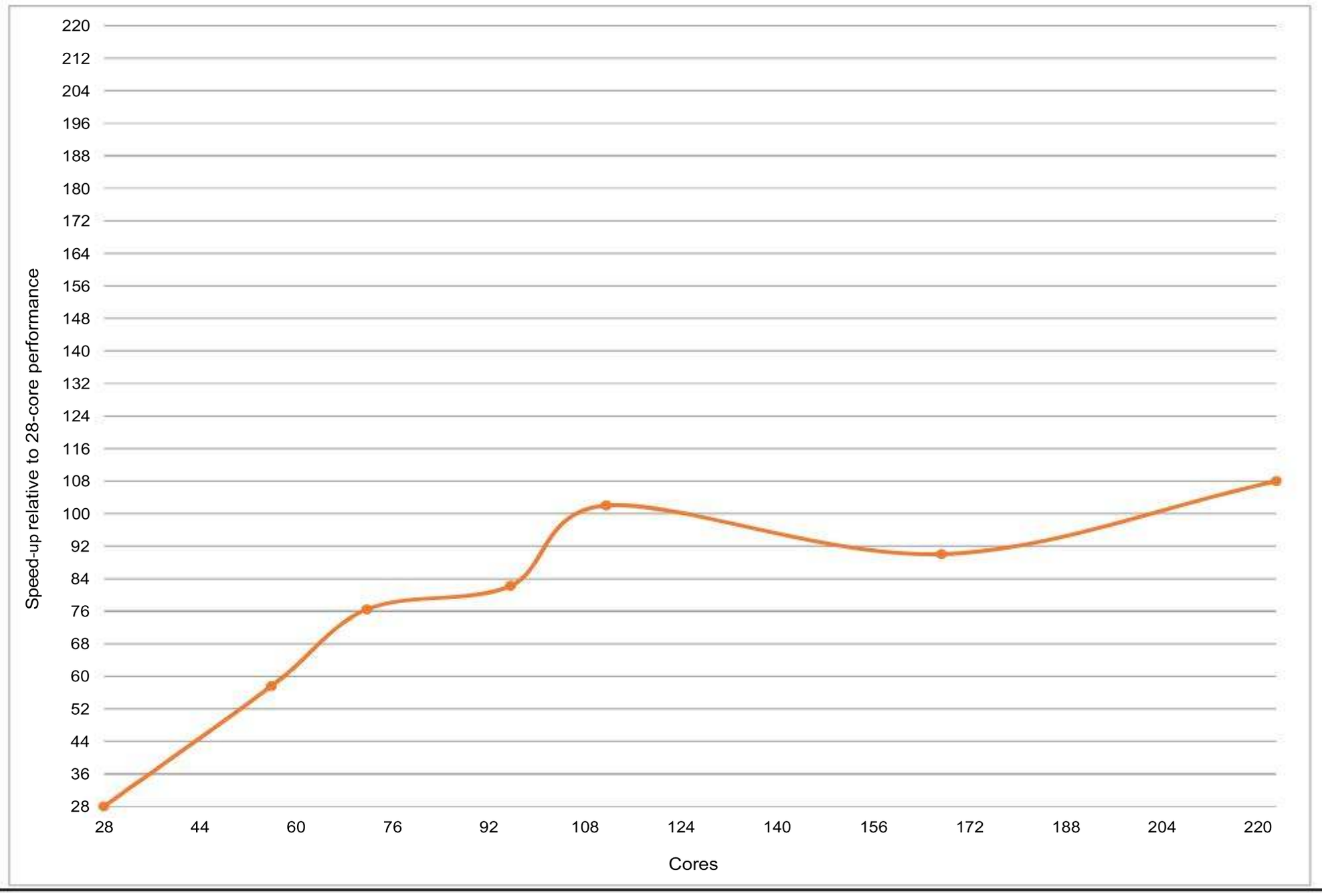


Figure 5.29 Performance of Intel Xeon Platinum systems from 28 cores to 224 cores on the SPECOMP2012 benchmark. Performance is relative to the 28-core system and shows good scaling at 56 and 72 cores before hitting performance limits.

the power consumed at full load, leading to a relative energy efficiency of about 85%.

In examining the SPECpower benchmarks, keep in mind that the benchmark consists of many independent tasks and that the number of tasks scales with the size of the processor. Hence the performance improvements obtained will be much better than those obtained when either the parallel tasks have more interaction or when the workload is not scaled. For example, consider the scaling for SPECintRate, which achieves between 60% and 70% of ideal scaling, despite being largely independent tasks, or for SPECOMP, which achieves about 50% efficiency in scaling to large core counts. The issue of scaling and its impact on both applications and performance is explored in more detail in Appendix I.

5.9 Fallacies and Pitfalls

Given our current understanding of parallel computing, there are many hidden pitfalls that will be uncovered either by careful designers or by unfortunate

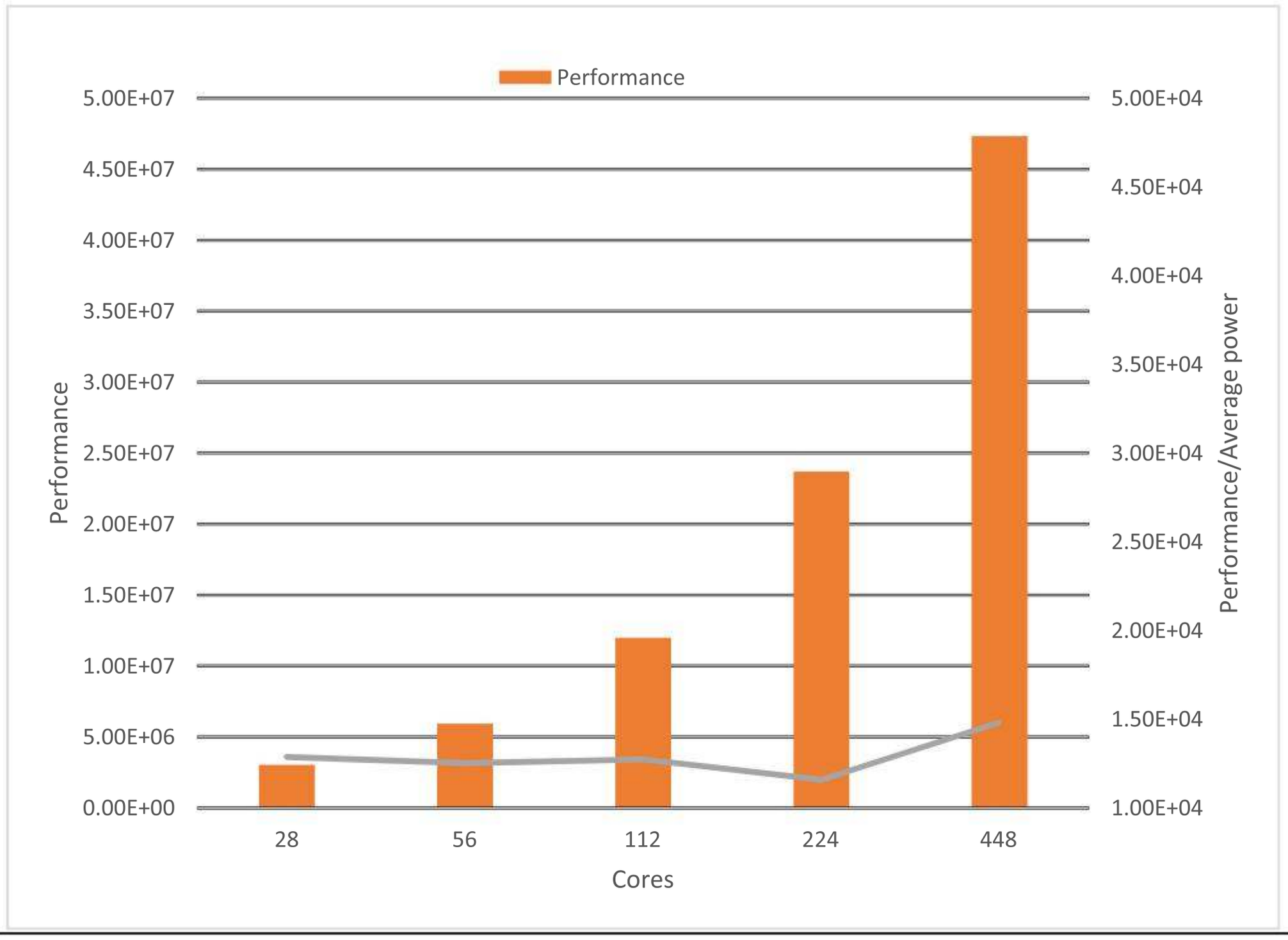


Figure 5.30 The performance and energy efficiency versus processor count for a Xenon Platinum running the SPECpower benchmark. The throughput is measured in ss_jops for the maximum workload and is on the left axis. Compared to the 28-core version, the 56, 112, 224, and 448 core versions show 99% to 100% efficiency in throughput. The line plots the energy efficiency with the units on the right axis. The energy efficiency varies by only a few percent across the range of core counts. Gray: Performance/Average Power.

ones. Given the large amount of hype that has surrounded multiprocessors over the years, common fallacies abound. We have included a selection of them.

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

Graphs like those in Figures 5.28 and 5.29, which plot performance versus number of processors, showing linear speedup, a plateau, and then a falling off, have long been used to judge the success of parallel processors. Although speedup is one facet of a parallel program, it is not a direct measure of performance. The first issue is the power of the processors being scaled: a program that linearly improves performance to equal 100 Intel Atom processors (the low-end processor used for

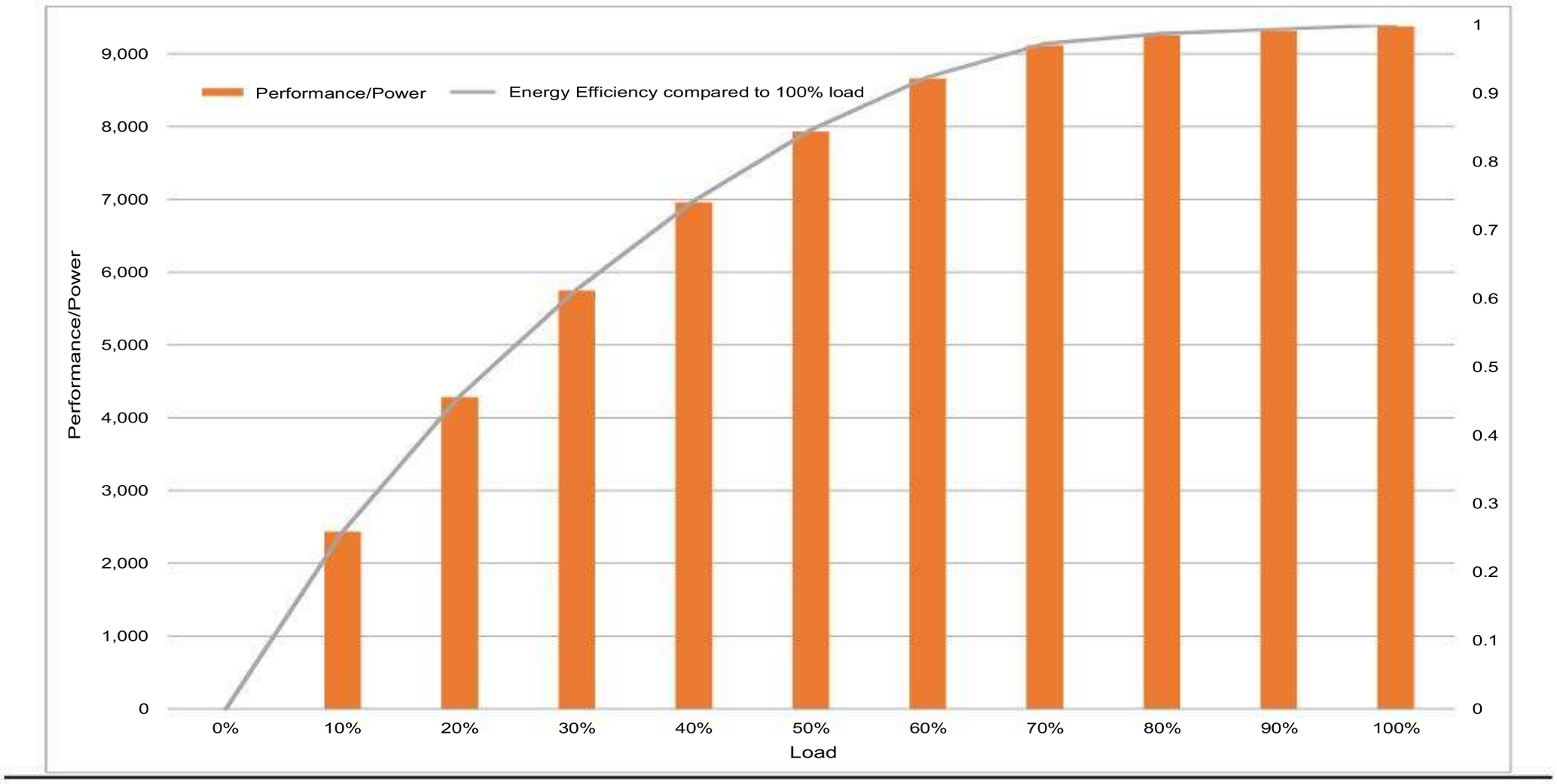


Figure 5.31 Performance/power and efficiency compared to a fully loaded system are shown various loads running on a 224-core Xeon Platinum. The columns plot the performance/power on the left axis, while the line shows the relative efficiency (performance/power) compared to a fully loaded system. Systems with 30% or less of the peak load are less than 65% as energy efficient as a fully loaded system, while those with loads of 70% or more are at least 97% as energy efficient as a fully loaded system.

netbooks) may be slower than the version run on an 8-core Xeon. Be especially careful of floating-point-intensive programs; processing elements without hardware assist may scale wonderfully but have poor collective performance. Likewise, benchmarks like SPECpower that have lots of independent parallelism and scale the problem with the multiprocessor size can achieve high efficiencies. Comparing execution times is fair only if you are comparing the best algorithms on each computer. Comparing the identical code on two computers may seem fair, but it is not; the parallel program may be slower on a uniprocessor than on a sequential version. Developing a parallel program will sometimes lead to algorithmic improvements, so comparing the previously best-known sequential program with the parallel code—which seems fair—will not compare equivalent algorithms. To reflect this issue, the terms *relative speedup* (same program) and *true speedup* (best program) are sometimes used.

Results that suggest *superlinear* performance, when a program on n processors is more than n times faster than the equivalent uniprocessor, may indicate that the comparison is unfair, although there are instances where "real" superlinear speedups have been encountered. For example, some scientific applications regularly achieve superlinear speedup for small increases in processor count (2 or 4 to 8 or 16). These results often arise because critical data structures that do not fit into the aggregate caches of a multiprocessor with 2 or 4 processors fit into the aggregate cache of a multiprocessor with 8 or 16 processors.

In summary, comparing performance by comparing speedups is at best tricky and at worst misleading. Comparing the speedups for two different multiprocessors does not necessarily tell us anything about the relative performance of the multiprocessors, as we also saw in the previous section. Even comparing two different algorithms on the same multiprocessor is tricky because we must use true speedup, rather than relative speedup, to obtain a valid comparison.

Pitfall *Amdahl's law doesn't apply to parallel computers.*

In 1987 the head of a research organization claimed that Amdahl's law (see Section 1.9) had been broken by an MIMD multiprocessor. This statement hardly meant, however, that the law has been overturned for parallel computers; the neglected portion of the program will still limit performance. To understand the basis of the media reports, let's see what Amdahl (1967) originally said:

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

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

The basis for the statement that Amdahl's law had been "overcome" was the use of *scaled speedup*, also called *weak scaling*. The researchers scaled the benchmark to have a dataset size that was 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark. For this particular algorithm, the sequential portion of the program was constant independent of the size of the input, and the rest was fully parallel—thus linear speedup with 1000 processors. Because the running time grew faster than linear, the program actually ran longer after scaling, even with 1000 processors.

Speedup that assumes scaling of the input is not the same as true speedup, and reporting it as if it were is misleading. Because parallel benchmarks are often run on different-sized multiprocessors, it is important to specify what type of application scaling is permissible and how that scaling should be done. Although simply scaling the data size with processor count is rarely appropriate, assuming a fixed problem size for a much larger processor count (called *strong scaling*) is often inappropriate, as well, because it is likely that users given a much larger multiprocessor would opt to run a larger or more detailed version of an application. See Appendix I for more discussion on this important topic.

pitfall *Linear speedups are needed to make multiprocessors cost-effective.*

It is widely recognized that one of the major benefits of parallel computing is to offer a "shorter time to solution" than the fastest uniprocessor. Many people, however, also hold the view that parallel processors cannot be as cost-effective as

uniprocessors unless they can achieve perfect linear speedup. This argument says that because the cost of the multiprocessor is a linear function of the number of processors, anything less than linear speedup means that the performance/cost ratio decreases, making a parallel processor less cost-effective than using a uniprocessor.

The problem with this argument is that cost not only is a function of processor count but also depends on memory, I/O, and the overhead of the system (box, power supply, interconnect, etc.). It also makes less sense in the multicore era, when there are multiple processors per chip.

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

Figure 5.32 shows the speedup for TPC-C, SPECintRate, and SPECfpRate on an IBM eServer p5 multiprocessor configured with 4–64 processors. The figure shows that only TPC-C achieves better than linear speedup. For SPECintRate and SPECfpRate, speedup is less than linear, but so is the cost, because unlike TPC-C, the amount of main memory and disk required both scale less than linearly.

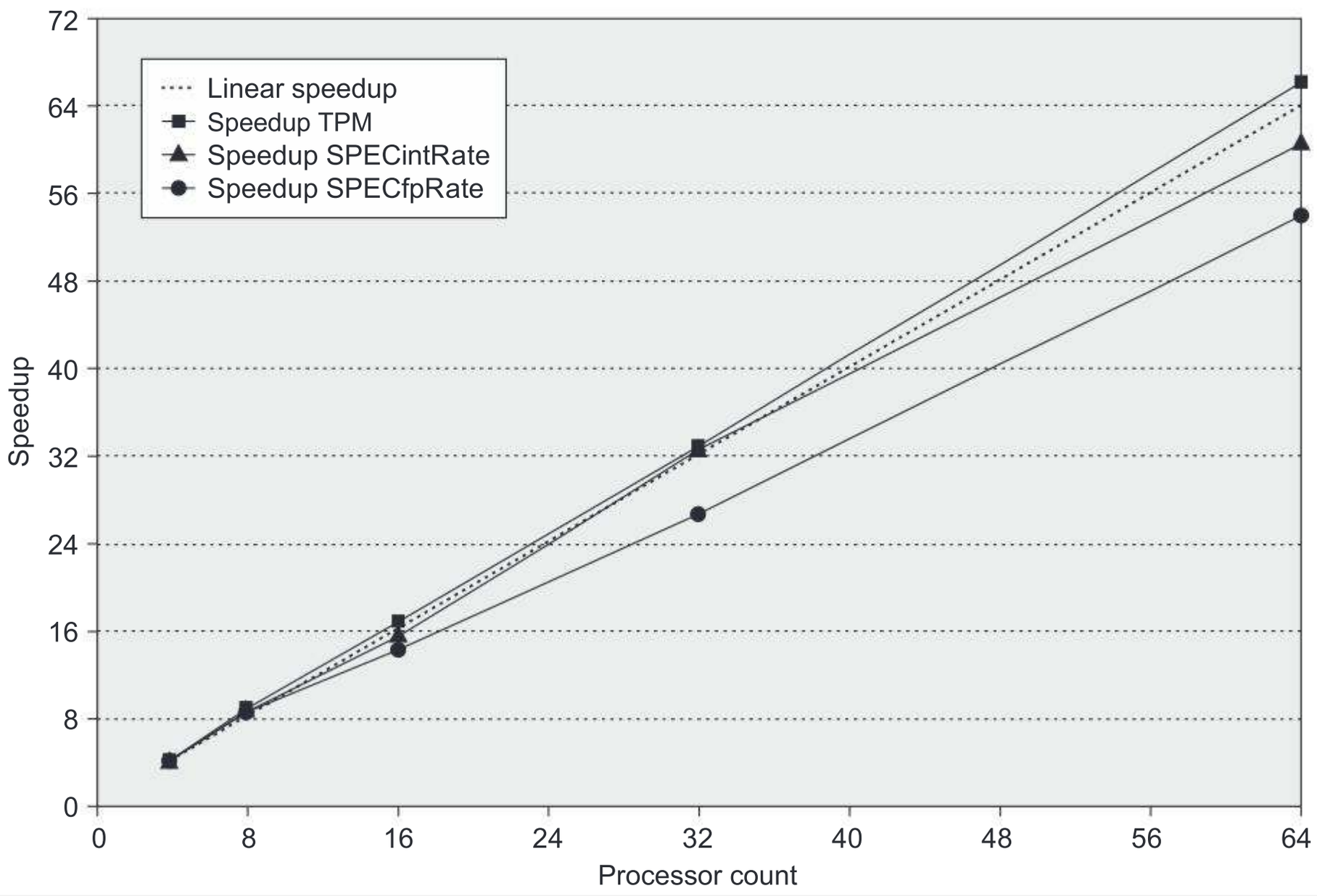


Figure 5.32 Speedup for three benchmarks on an IBM eServer p5 multiprocessor when configured with 4, 8, 16, 32, and 64 processors. The *dashed line* shows linear speedup.

As Figure 5.33 shows, larger processor counts can be more cost-effective than the 4-processor configuration. In comparing the cost-performance of two computers, we must be sure to include accurate assessments of both total system cost and what performance is achievable. For many applications with larger memory demands, such a comparison can dramatically increase the attractiveness of using a multiprocessor.

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

There is a long history of software for multiprocessors lagging the development of uniprocessor software, probably because the problems are much harder. We give

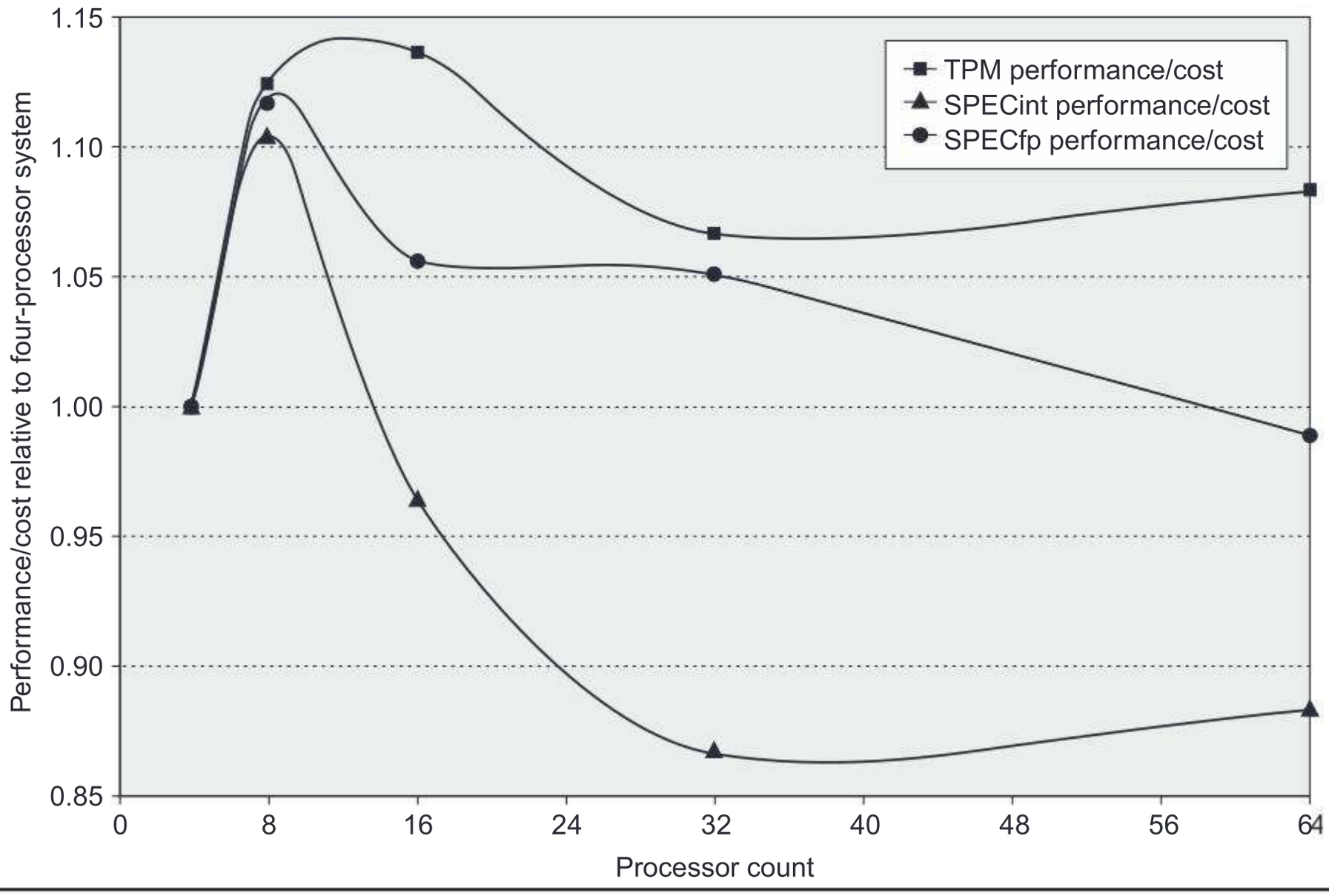


Figure 5.33 The performance/cost for IBM eServer p5 multiprocessors with 4—64 processors is shown relative to the 4-processor configuration. Any measurement above 1.0 indicates that the configuration is more cost-effective than the 4-processor system. The 8-processor configurations show an advantage for all three benchmarks, whereas two of the three benchmarks show a cost-performance advantage in the 16- and 32-processor configurations. For TPC-C, the configurations are those used in the official runs, which means that disk and memory scale nearly linearly with processor count, and a 64-processor machine is approximately twice as expensive as a 32-processor version. In contrast, the disk and memory are scaled more slowly (although still faster than necessary to achieve the best SPECrate at 64 processors). In particular, the disk configurations go from one drive for the 4-processor version to four drives (140 GB) for the 64-processor version. Memory is scaled from 8 GiB for the 4-processor system to 20 GiB for the 64-processor system.

one example to show the subtlety of the issues, but there are many examples we could choose from.

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the SGI operating system in 2000 originally protected the page table data structure with a single lock, assuming that page allocation was infrequent. In a uniprocessor this does not represent a performance problem. In a multiprocessor it can become a major performance bottleneck for some programs.

Consider a program that uses many pages that are initialized at startup, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table data structure, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time).

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

As multicore became the dominant theme in everything from desktops to servers, the lack of an adequate investment in parallel software became apparent. Given the lack of focus, it will likely be years before the software systems we use fully exploit the growing number of cores.

5.10 The Future of Multicore Scaling

For more than 40 years, researchers and designers have predicted the end of uniprocessors and their dominance by multiprocessors. Until the early years of this century, this prediction was constantly proven wrong. As we saw in Chapter 3, the costs of trying to find and exploit more ILP became prohibitive in efficiency (both in silicon area and in power). Of course, multicore does not magically solve the power problem because it clearly increases both the transistor count and the active number of transistors switching, which are the two dominant contributions

to power. As we will see in this section, energy issues have already and are likely to limit multicore scaling more severely than previously thought.

ILP scaling failed because of both limitations in the ILP available and the efficiency of exploiting that ILP. Similarly, a combination of two factors means that simply scaling performance by adding cores is unlikely to be broadly successful, at least as a method of making individual programs run faster. This combination arises from the challenges posed by Amdahl's law, which assesses the efficiency of exploiting parallelism, and the end of Dennard's scaling, which dictates the energy required for a multicore processor.

To understand these factors, we take a simple model of both technology scaling (based on an extensive and highly detailed analysis in Esmaeilzadeh et al., (2012)). Let's start by reviewing energy consumption and power in CMOS. Recall from Chapter 1 that the energy to switch a transistor is given as

$$\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$$

CMOS scaling is limited primarily by thermal power, which is a combination of static leakage power and dynamic power, which tends to dominate. Power is given by

$$\begin{aligned}\text{Power} &= \text{Energy per Transistor} \times \text{Frequency} \times \text{Transistors switched} \\ &= \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency} \times \text{Transistors switched}\end{aligned}$$

To understand the implications of how energy and power scale, let's compare the technology used in an earlier Xeon E5 (2014/2015) using 22 nm technology with the 10 nm technology used in a Xeon Platinum (2021). Figure 5.34 shows this comparison based on technology projections and resulting effects on energy and power scaling. Notice that power scaling >1.0 means that the future device consumes more power, in this case, 1.79× as much.

Consider the implications of this for one of the latest Intel Xeon Platinum 8380 processor, which has 40 cores, close to 10 billion transistors (including

Device count scaling (since a transistor is 1/4 the size)	4
Frequency scaling (based on projections of device speed)	1.75
Voltage scaling projected	0.81
Capacitance scaling projected	0.39
Energy per switched transistor scaling (CV^2)	0.26
Power scaling assuming fraction of transistors switching is the same and chip exhibits full frequency scaling	1.79

Figure 5.34 A comparison of the 22 nm technology of 2014/5 with a current 11 nm technology, used by Intel in the 2021 Xeon Platinum. The characteristics are from the International Technology Roadmap for Semiconductors, which for many years set the course for future generations of semiconductor technology. With the challenges in continued scaling and the resulting uncertainty about new technologies, the Roadmap was halted after the 2015 edition. This comparison shows over a six-year period the impact of the slowdown in Dennard scaling, as well as the slowdown in Moore's law, which would have led to the availability of 11 nm in 2018/9.

60 MiB of cache), operates at 2.3 GHz, has a thermal power rating of 270 watts, and has a die size of 600 mm^2. The clock frequency is already limited by power dissipation: an 8-core version (Xeon Gold) has a clock of 3.2 GHz and dissipates 140 W. Comparing versions of Xeon's built between 2017 and 2022 shows that clock rates get faster (by about 50%) if core counts are not increased and remain largely the same with an increase in core counts. Power dissipation is the key limiter. Furthermore, we may have Amdahl's law effects, as the next example shows.

Example Suppose we have a 96-core future-generation processor, but on average, only 54 cores can be busy. Suppose that 90% of the time, we can use all available cores; 9% of the time, we can use 50 cores; and 1% of the time is strictly serial. How much speedup might we expect? Assume that cores can be turned off when not in use and draw no power and assume that the use of a different number of cores is distributed so that we need to worry only about average power consumption. How would the multicore speedup compare to the 24-processor count version that can use all its processors 99% of the time?

Answer We can find how many cores can be used for the 90% of the time when more than 54 are usable, as follows:

$$\begin{aligned}\text{Average Processor Usage} &= 0.09 \times 50 + 0.01 \times 1 + 0.90 \times \text{Max processor}\\ 54 &= 4.51 + 0.90 \times \text{Max processor}\\ \text{Max processor} &= 55\end{aligned}$$

Now, we can find the speedup:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{55}}{55} + \frac{\text{Fraction}_{50}}{50} + (1 - \text{Fraction}_{55} - \text{Fraction}_{50})}$$

$$\text{Speedup} = \frac{1}{\frac{0.90}{55} + \frac{0.09}{50} + 0.01} = 35.5$$

Now compute the speedup on 24 processors:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{24}}{24} + (1 - \text{Fraction}_{24})}$$

$$\text{Speedup} = \frac{1}{\frac{0.99}{24} + 0.01} = 19.5$$

When considering both power constraints and Amdahl's law effects, the 96-processor version achieves less than a factor of 2 speedup over the 24-processor version. In fact, the speedup from clock rate increase nearly matches the speedup from the 4× processor count increase. We comment on these issues further in the concluding remarks.

The Future of Multicore Is Heterogeneity

Straight scaling of multicores is likely to encounter the power wall we have just discussed. The future of multicore is toward more heterogeneous systems. For PMDs and desktops, these are likely to be system-on-chip (SoC) designs containing multiple general-purpose cores, as well as domain-specific accelerators. The Apple M1 and M2 are good examples of such systems, and we will examine the M1 design shortly. For large-scale multicores used in data centers, off-chip accelerators using domain-specific architectures are an increasing trend. By implementing the most intensive portions of a computation, such as a neural network training or inference, in a more energy-efficient fashion, the general-purpose processors inside the multicore can be used more efficiently. Whether multicores for data center applications will incorporate different types of general-purpose cores is an open issue at present, but the limitation of power dissipation may lead in that direction.

The Apple M1: A Heterogeneous, SoC Multicore

The M1, introduced in late 2020, contains the following:

- General-purpose ARM cores, four of which are high performance (and higher power) and four are efficiency cores.
- An 8-core GPU capable of 2.6 teraops performance for graphics tasks.
- A 16-core neural engine that can speed up machine-learning inference tasks by 10–15 times versus the general-purpose cores, at lower power consumption.
- Signal processing engines to handle image processing functions and media encode/decode.
- An encryption engine to support security enclaves.

For many applications, the M1 provides better performance at lower power. About a year after the Apple M1, Intel introduced a variety of new processors that include some of the same capabilities, specifically the use of performance and efficiency cores. While Apple has not published data on the relative performance and power of the two core types, measurements on Intel designs using performance and efficiency cores (available in the Intel i9 series) show that the performance cores are about 1.5 times faster without using SMT, which is only included in the performance cores. With a parallel workload that can use the SMT capability, the performance core is 1.8–2.0 times faster. The power consumption of the performance cores is 1.3–1.7 times higher, depending on whether SMT is in use.

5.11 Concluding Remarks

As we saw in the previous section, multicore does not magically solve the power problem because it clearly increases both the transistor count and the active number of transistors switching, which are the two dominant contributions to power. The slowdown of Dennard scaling merely makes it more extreme.

But multicore does alter the game. By allowing idle cores to be placed in power-saving mode and incorporating cores with different performance/power characteristics, an improvement in power efficiency can be achieved, as the results in this chapter have shown. For example, shutting down cores in the Intel Xeon or i9 allows other cores to operate in Turbo mode. This capability allows a trade-off between higher clock rates with fewer processors and more processors with lower clock rates. Likewise, using the M1's efficiency core for less demanding tasks reduces power consumption and enhances battery life. Similarly, an SoC design incorporates specialized processors or accelerators that perform key functions at both higher performance and lower power compared to a general-purpose core.

Multicore also shifts the burden for keeping the processor busy by relying more on thread-level and request-level parallelism, which the application and programmer are responsible for identifying, rather than on ILP, for which the hardware is responsible. Multiprogrammed and highly parallel workloads that avoid Amdahl's law effects will benefit more easily. Similarly, domain-specific accelerators, such as GPUs or deep-learning processors (see Chapter 7), can overcome bottlenecks by exploiting more efficient organizations (SIMD vs. MIMD or program-controlled memories vs. caches), as well as by exposing more parallelism.

In the edition published in 2012 we raised the question of whether it would be worthwhile to consider heterogeneous processors. At that time, no such multicore was delivered or announced, and heterogeneous multiprocessors had seen only limited success in special-purpose computers or embedded systems. Combining domain-specific processors, like those discussed in Chapters 4 and 7, with general-purpose processors is perhaps the best road forward to achieve increased performance and energy efficiency while maintaining some of the flexibility that general-purpose processors offer.

5.12 Historical Perspectives and References

Section M.7 (available online) looks at the history of multiprocessors and parallel processing. Divided by both time period and architecture, the section features discussions on early experimental multiprocessors and some of the great debates in parallel processing. Recent advances are also covered. References for further reading are included.

Case Studies and Exercises by Amr Zaky

Case Study 1: Single Chip Multicore Multiprocessor

Concepts illustrated by this case study.

- Snooping Coherence Protocols Transitions/Actions
- Snooping Coherence Protocols Performance
- Snooping Coherence Protocols Implementation Optimization
- Synchronization and Its Use of Cache Coherence Support

Transaction Notation

The following notation is used in the case study.

Pi, r/w, Mj → processor core i reads/writes memory block #j. For example,

P2, r, M3: processor core 2 reads memory block #3.

Caches Configuration/Policy

The caches used in the case study (L1 and L2) are all direct-mapped, write-back, write-allocate caches. An L2 cache is assumed *inclusive* of the L1 caches connected to it. **The line sizes of all the L1 caches and L2 cache(s) are the same and are equal to the size of a memory block.**

Coherence Protocol

The coherence protocol employed is a write-invalidate snooping protocol with the states M, S, and I, as in Section 5.3.

Memory Distribution and Addressing

5.1. [20] <5.3> Figure 5.35 depicts a 4-processor SMP. Each processor core has a private L1 cache, and all cores share an L2 cache. The caches' configurations and

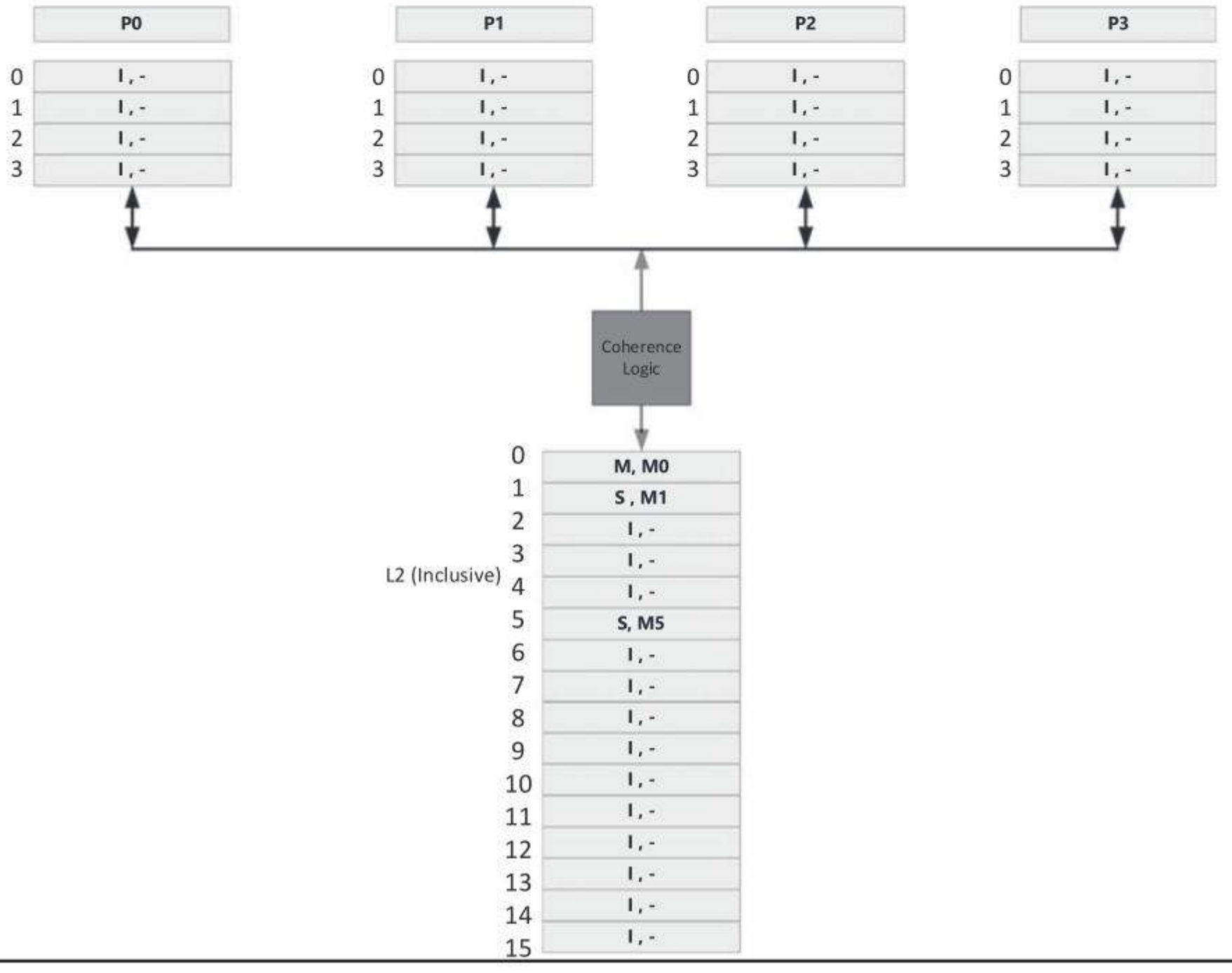


Figure 5.35 Structure of four-processor SMP.

policies are described in the case study introduction. Consult the figure for the initial states, and data of L2 cache lines. Mi stored in an L2 block entry denotes the entry is caching memory block i,

- Describe the changes to the cache line states and the actions taken by the caches and by the snooping logic as the following sequence of memory reads and writes progresses. In the sequence, x | | y means operation x is followed by operation y. Mi refers to physical memory block i.

```
<P0, r, M0 || P3, w, M1 || P1, r, M5 || P2, r, M5 || P0, r, M0 || P3, r,
M1|| P2, w, M5>
```

5.2. [30] <5.3> Figure 5.36 shows the different delays (in processor cycles) associated with an implementation of the snooping protocol used.

Access	Processor Cycles	Uses
L1 tag access	1	L1 read/write hit, first cycle of miss, snoop query from coherence logic
L1 data access	1	L1 read (concurrent with tag access), L1 write
L2 tag access	6	L1 miss
L2 data access	12	L1 writeback, L1 refill
Coherence logic access	1	L1 miss, L1 upgrade miss (invalidate), L1 response to coherence logic.

The tag access and the read data access are done concurrently. If the read misses then the data is discarded. On the other hand, the write should not access the data array until the tag accesses confirms the write is a hit. So a write hit takes two cycles. The underlying assumption is the clock cycle is too short to accommodate a tag access serially followed by a data access.

Figure 5.36 Latencies for snooping protocol actions.

Example:

The latencies for a P1 write miss for a memory block which is modified in P2's L1 cache are shown in Figure 5.37.

Step	Delay in cycles
P1 tag access	1 cycle
P1 (miss) invalidate request	1 cycle
Coherence snoop query to P2 (and all cores) tags	1 cycle
P2 write-back to L2	12 cycles
L2 refill to P1	12 cycles
Total	27 cycles

Figure 5.37 Write miss latencies.

We assume the processors on the bus will announce their status (L1 hit shared line, L1 hit modified line, L1 miss) with respect to the invalidated line in one cycle.

- Calculate the total number of delay cycles for the read/write sequence of question 5.1.

5.3 [10/10] <5.3> If the cache line size is such that it holds one element of array A, and if the array A is totally in L2, but not in P0's L1 cache.

```
for (…) A[i] = A[i]+1;
```

- How many cycles will the two memory accesses in every iteration of the for loop above take?
- How many cycles will the memory accesses take if the protocol is changed from MSI, to MESI?

5.4 [15] <5.3> To investigate the performance of MSI and MESI protocols further, explain why the following two loops' data accesses will take different number of cycles when the MSI protocol is used. (Assume A[i] is not in the L1 cache initially but array A is entirely in L2), while the number of cycles should not vary significantly when the MESI protocol is used.

```
for (...) A[i] = A[i-1] +1;
   for (...) A[i] = A[i]+1;
```

5.5 [15/15] <5.3> Consider the following sequence of memory reads and writes.

```
<P0, r, M0 || P3, w, M1 || P1, r, M5 || P2, r, M5 || P2, r, M0 || P0, r,
M1|| P2, w, M1>
```

- Assume none of the memory blocks accessed is in the L1 cache and all of them are in the L2 cache.
  - Calculate the total number of delay cycles for the read/write sequence above.
  - Calculate the total number of delay cycles, if a processor core can supply another with the data rather than waiting for L2 to supply it. Assume a data transfer from one processor L1 to another L1 cache takes 4 cycles.

An alternative L2 cache structure is used as shown in Figure 5.38. Every two cores share a private L2 cache. The private L2 cache is **inclusive** of the L1 caches of the core it is attached to. We also assume the LLC to be **inclusive** of the 2 L2 caches. A cache line miss in L1 will be serviced from L2 unless L2 it misses in L2 too. In that case the cache line will be fetched from the LLC (which might have to fetch it from the memory). The cache line fetched from the LLC will be first written in L2, then passed on to the requesting L1. In the presence of coherence logic, this process will usually require coherence actions and possible changes to the status of the cache line if it was present in any other cache (L1, L2, or LLC).

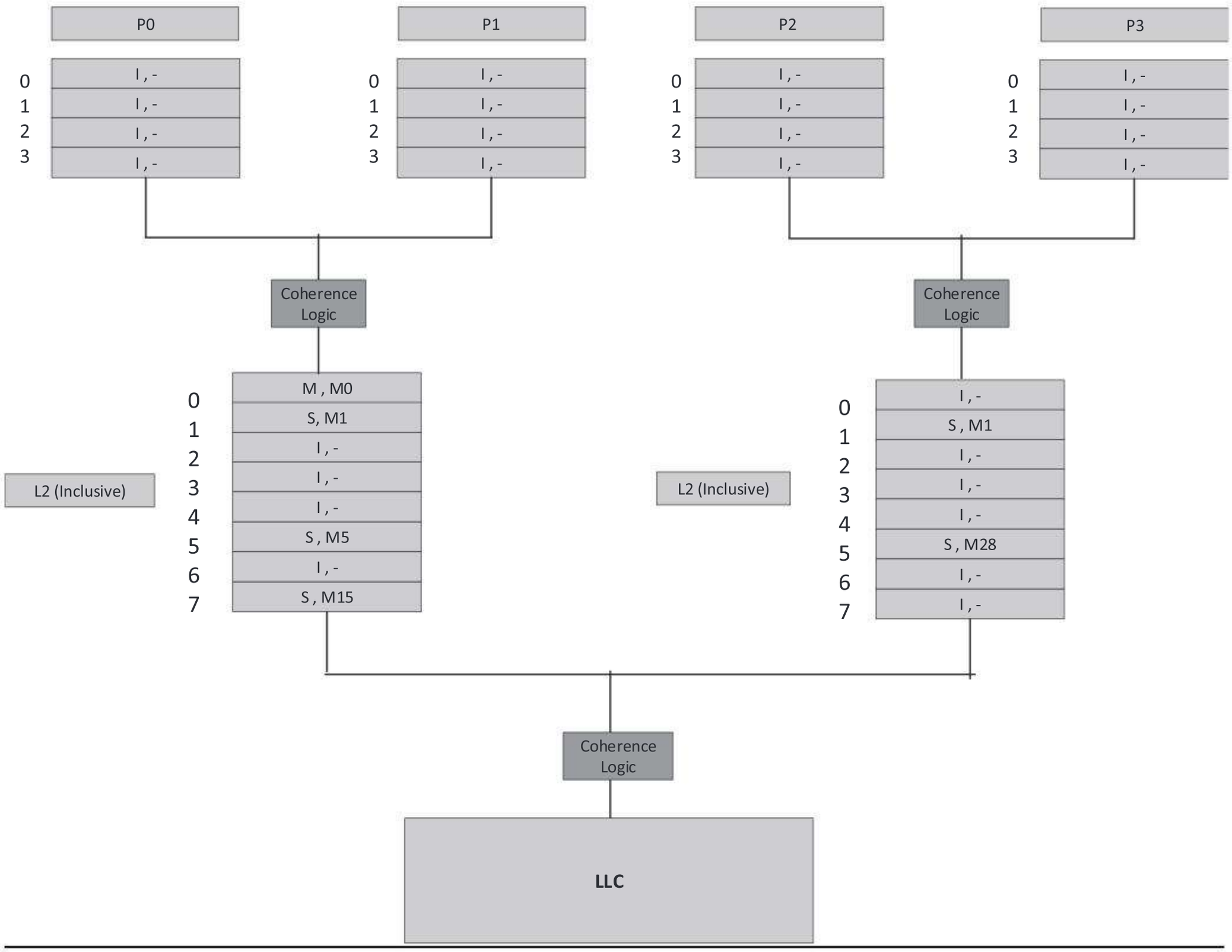


Figure 5.38 Four-processor design with Shared L2.

5.6 [20] <5.3> Address the snooping protocol implementation for this new configuration. For example, what states will P0 L1 and (P0/P1) L2 caches have after a cache line miss (in both L1 and L2) to memory block M38? While we might be tempted to ignore L1 coherence logic (between L1 and L2) altogether and exclusively use L2 coherence logic (between L2 and LLC), implementing L1/L2 coherence logic will provide some performance optimization opportunities. Show an example.

5.7 **[10] <5.3>** Explain why combining both the MESI protocol in the L2 caches and a write-once protocol in the L1 cache reduces the amount of coherence traffic for this configuration. In a write-once L1 cache policy the cache acts as a write-through cache for the first write to a cache line, and as a writeback cache for ensuing writes.

5.8 **[20] <5.3>** Calculate the total number of read/write delay cycles for the following sequence of memory accesses. Assume the MSI protocol is used.

```
   <P0, r, M0 || P3, w, M1 || P1, r, M5 || P2, r, M5 || P1, r, M0 || P0, r,
M1|| P2, w, M1>
```

5.9 **[20] <5.5>** The code below depicts the two implementations of a spin lock loop discussed in section 5.5.

```
lockit: addi x2, x0, #1
        EXCH x2,0(x1)
        bnez x2, lockit
```

```
lockit: ld x2, 0(x1)
        bnez x2, lockit
        bnez x2, lockit
        addi x2, x0, #1
        EXCH x2,0(x1)
```

- Show the spin lock loop timing for both implementations when P0 and P1 are competing for a lock held by P2.

Case Study 2: Simple Directory-Based Coherence

Concepts illustrated by this case study.

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

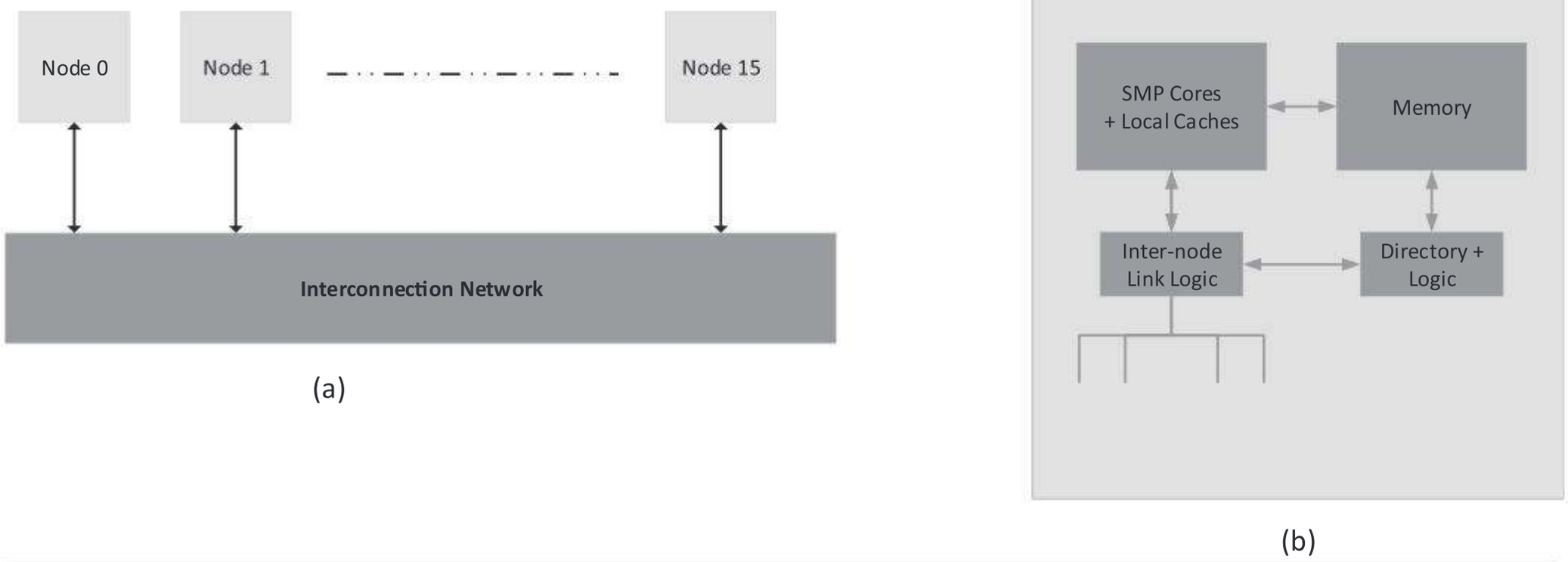


Figure 5.39 Distributedshared-memorydesign with 16 nodes (a). Each node is organized as shown in (b). Consider the distributed shared-memory system illustrated in Figure 5.39 (a). It consists of 16 nodes of processor cores connected to an interconnection network IN. Figure 5.39 (b) depicts the internal structure of a node, and while it conveys the possibility of every node having multiple cores (connected as an SMP), for starters, one can assume that every node has a single core. However, if the nodes have multiple cores, then it can be assumed that the cores use snooping to address coherency amongst themselves.

Coherence Protocol

The coherence protocol employed is a write-invalidate directory-based protocol with the states M, S, and I, as in Section 5.4.

Memory Distribution and Addressing

Memory is distributed equally among nodes. The most significant 4 bits of a memory block address represent the node number at which the memory is located. Unless otherwise specified in a problem, the size of a memory block and the number of blocks in a distributed memory are unspecified.

Transaction Notation

The following notation is used in the case study.

Ni, r/w, M (m, n)→ processor: Node i reads/writes memory block n at node m. For example,

N2, r, M (3,7): Node 2 reads memory block #7 at node 3.

Caches Configuration/Policy

The caches used in the case study (L1, L2, …) are all direct-mapped, write-back, write-allocate caches. A cache is assumed *inclusive* of the caches above it (closer to the processor). **The line sizes of all the caches are the same and are equal to the size of a memory block.**

5.10 **[20] <5.4>** Assuming all cache blocks are initially in the invalid state, describe the changes to the cache line states, the actions taken by the directory controllers, and the coherence messages exchanged as the following sequence of memory accesses progresses. In the sequence, x | | y means operation x is followed by operation y.

```
<N0, r, M (0, 4) || N3, w, M (1, 5) || N7, r, M (1,5) || N2, r, M (5,2)
|| N0, w, M (0,4) || N13, r, M (1,5) || N2, w, M(5,2)>
```

5.11 [5/10] <5.4> If every node is allocated 1 GiB of distributed memory, and the memory block size is 128 Bytes:

a. Describe a straightforward layout of a node directory data structure and calculate the total number of bits needed to implement it. Also, calculate the directory size as a percentage of the memory size.
b. Using the layout suggested in part (a), sketch the directory contents for node 1. As a reminder, all accesses that target memory blocks M(1, *) access node 1's directory.

5.12 [5/5/5] <5.4> To reduce the directory size either or both these changes to its structure can be made:

- Reduce the number of entries in each directory by having every directory entry account for the coherence state of a chunk of memory larger than a block.
- Reduce the size of each entry by having a directory entry refer to memory usage by multiple nodes instead of individual nodes.

a. Calculate the number of entries in a directory data structure when each entry is accounting for the usage of 4KiB of memory (every node is allocated 1 GiB of distributed memory).
b. Calculate the size of one directory entry if 16 nodes of the system are divided into 4 groups (every group has 4 nodes) and the entry refers to ownership of a memory block by a group of nodes rather than individual nodes.
c. Calculate the size of the directory if the above two optimizations are simultaneously employed.

5.13 [15/5] <5.4> Considering the straightforward directory implementation of case study problem 5.11, sketch the set of coherence messages needed to carry out the MSI protocol. Provide enough details specifying addresses, source/destination nodes, data, and commands.

a. Sketch the different fields of each message type and calculate the size of each message type in bits.
b. Repeat (a) while padding the messages with extra bits - if needed - to make the message size a byte multiple.

5.14 [10/10/10/10/5] <5.4> The directory size optimizations suggested in problem 5.12 come at a cost in the number of messages needed to enforce the directory-based coherence protocol.

Use the following sequence of memory accesses to compare the number of messages exchanged under different optimizations.

```
<N0, r, M (0, 4) || N3, w, M (0, 5) || N7, r, M (1,60) || N5, r, M
(1,59) || N0, w, M (0,5)>
```

a. Describe the actions taken and coherence messages exchanged when no directory size optimization exists (as in 5.11).
b. Describe the actions taken and coherence messages exchanged when the change in 5.12 (a) is implemented (every directory entry manages a 4KiB memory block).
c. Describe the actions taken and coherence messages exchanged when the change in 5.12 (b) is implemented (nodes are arranged as four groups: (G0:

N0, N1, N2, N3), (G1: N4, N5, N6, N7), (G2: N8, N9, N10, N11), (G3: N12, N13, N14, N15)).

d. Describe the actions taken and coherence messages exchanged when the changes in both 5.12 (a) and (b) are implemented (nodes are arranged as four groups: (G0: N0, N1, N2, N3), (G1: N4, N5, N6, N7), (G2: N8, N9, N10, N11), (G3: N12, N13, N14, N15)).

e. Fill in the table below showing the directory size per node (bits) and the number of messages exchanged for the above sequence of transactions, as a function of number of nodes in a coherence group and block size managed by a directory entry.

Number of nodes in coherence group	Directory size per node (bits)	Number of messages exchanged for the above sequence of transaction.
	Memory block size managed per entry	
1	128 Bytes	
	4 KiB	
4	128 Bytes	
	4 KiB	

5.15 [10] <5.4> Stretching the optimization in Exercise 5.12 (c) to its logical extreme, the node directory can be eliminated (no record of what node is using what memory block will be kept). Assuming some of the directory logic is kept, this organization is very demanding in the number of coherence messages exchanged. As expensive in the number of messages as this organization is, would there still be any value for using a directory logic? Hint: Assume that messages from any node i to any node j follow a deterministic path in the interconnection network and are dealt with in-order, and assume nodes i and j need to access (read or write) a memory block in node k.

Case Study 3: Memory Consistency

Concepts illustrated by this case study.

- Legitimate Program Behavior Under Sequential Consistency (SC) Models
- Hardware Optimizations Allowed for SC Models
- Using Synchronization Primitives to Make a Consistency Model Emulate a More Restrictive Model

5.16 [10/5/5/] <5.6> The following code segment depicts Peterson's mutual exclusion algorithm for two processes running on two processors. The algorithm was designed to demonstrate the possibility of enforcing mutual exclusion without using hardware synchronization primitives.

Initial condition: enter0, enter1= false/*initially both processes P0 and P1 do not wish to enter the critical section. */

```
P0: enter0 ← true
 turn ← 0
 wait-for ((not enter1) OR turn == 1)
 begin
/* critical section */
 end
 enter 0← false
```

```
P1: enter1 ← true
 turn ← 1
 wait-for ((not enter0) OR turn == 0)
 begin
/* critical section */
 end
 enter1 ← false
```

a. Show that when both processes want to enter the critical section and the processors conform to Sequential Consistency (SC) model, then only one process will enter the critical section at a time.
b. Show a SC ordering of the reads and writes of P0 and P1 that will allow P0 to enter the critical section before P1.
c. Show a SC ordering of the reads and writes of P0 and P1 that will allow P1 to enter the critical section before P0.

5.17 [10] <5.6>Show that the Peterson's algorithm is not guaranteed to behave correctly when the processors are implementing a Release Consistency (RC) model. Illustrate by showing some RC ordering that allows both processes P1 and P2 to enter the critical section at the same time.

5.18 [5/5/5] <5.6> Assume a and b are initially 0.

```
P0: a=1
 while (b == 0)
 print ("p0 done\n")

P1: b=1
 while (a == 0)
 print ("p1 done\n")
```

a. What are the possible outputs printed for the above processes when using a SC model?
b. Explain the possibility that the two processes might never pass the while loop when the code is executed—as is—on a system using RC model.
c. Show a compiler code reordering that might cause the same effect described in (b) on a system using SC model.

5.19 [10] <5.6> If P0 in problem 5.18 prefetches b to its cache before P1 sets it to 1, will this cause unexpected results on a system using SC model? Assume the presence of a cache coherence mechanism.

5.20 [10/10] <5.6> Augment the mutual exclusion code of problem 5.15 with synchronization primitives so that it behaves correctly on a system implementing the RC model.

5.21 [20/20/20] <5.6> Sequential consistency (SC) requires that all reads and writes appear to have executed in some total order. This may require the processor to stall in certain cases before committing a read or write instruction. Consider the following code sequence:

```
write A
read B
```

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

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

Assume that one memory operation can be performed per cycle and that operations that hit in the cache or that can be satisfied by the write buffer introduce no stall cycles. Operations that miss incur 100 cycles of memory latency.

How many stall cycles occur *prior* to each operation for both the SC and TSO consistency models? (Write buffer can hold at most one write). The memory accesses are described using the notation introduced in Exercise 5.1.

a.	[20] <5.6>	P0, W, M110	//assume miss (no other cache has the line)
		P0, R, M108	//assume miss (no other cache has the line)
b.	[20] <5.6>	P0, R, M110	//assume miss (no other cache has the line)
		P0, W, M100	// assume hit.
c.	[20] <5.6>	P0, W, M100	//assume miss.
		P0, W, M100	// assume hit.

Exercises

5.22 [15/20/10] <5.1> In this exercise we examine the effect of the interconnection network topology on the CPI of programs running on a 64-processor distributed-memory multiprocessor. The processor clock rate is 2.0 GHz, and the base CPI of an application with all references hitting in the cache is 0.75. Assume that 0.2% of the instructions involve a remote communication reference. The cost of a remote communication reference is (100 + 10 h) ns, h being the number of communication network hops that a remote reference has to make to the remote processor memory and back. Assume all communication links are bidirectional.

a. [15] <5.1> Calculate the worst-case remote communication cost when the 64 processors are arranged as a ring, as an 8×8 processor grid, or as a hypercube (Hint: Longest communication path on a 2^n hypercube has n links).
b. [20] <5.1> Compare the base CPI of the application with no remote communication to the CPI achieved with each of the three topologies in part (a).

5.23 [15] <5.2> Show how the basic snooping protocol of Figure 5.6 can be changed for a write-through cache. What is the major hardware functionality that is not needed with a write-through cache compared with a write-back cache?

5.24 [20/20] <5.2> Please answer the following problems:

a. [20] <5.2> Add a clean exclusive state to the basic snooping cache coherence protocol (Figure 5.6). Show the protocol in the finite state machine format used in the figure.
b. [20] <5.2> Add an "owned" state to the protocol of part (a) and describe using the same finite state machine format used in Figure 5.6.

5.25 [15] <5.2> One proposed solution for the problem of false sharing is to add a valid bit per word. This would allow the protocol to invalidate a word without removing the entire block, letting a processor keep a portion of a block in its cache while another processor writes a different portion of the block. What extra complications are introduced into the basic snooping cache coherence protocol (Figure 5.6) by this addition? Consider all possible protocol actions.

5.26 [15/20] <5.3> This exercise studies the impact of aggressive techniques to exploit instruction-level parallelism in the processor when used in the design of shared-memory multiprocessor systems. Consider two systems identical except for the processor. System A uses a processor with a simple single-issue, in-order pipeline, and system B uses a processor with four-way issue, out-of-order execution, and a reorder buffer with 64 entries.

a. [15] <5.3> Following the convention of Figure 5.11, let us divide the execution time into instruction execution, cache access, memory access, and other stalls. How would you expect each of these components to differ between system A and system B?
b. [10] <5.3> Based on the discussion of the behavior of OLTP workload in Section 5.3, what is the important difference between the OLTP workload and other benchmarks that limit benefit from a more aggressive processor design?

5.27 [15] <5.3> How would you change the code of an application to avoid false sharing? What might be done by a compiler and what might require programmer directives?

5.28 [15] <5.3> An application is calculating the number of occurrences of a certain word in a very large number of documents. A very large number of processors divided the work, searching the different documents. They created a huge array—

word_count—of 32-bit integers, every element of which is the number of times the word occurred in some document. In a second phase, the computation is moved to a small SMP server with four processors. Each processor sums up approximately ¼ of the array elements. Later, one processor calculates the total sum.

```
for (int p = 0; p<=3; p++) // Each iteration of is executed on a separate
processor.
{
sum [p] = 0;
for (int i = 0; i < n/4; i++) // n is size of word_count and is divisible
by 4
sum[p] = sum[p] + word_count[p+4*i];
}
   total_sum = sum[0] + sum[1]+sum[2]+sum[3] //executed only on
processor.
```

a. Assuming each processor has a 32-byte L1 data cache, identify the cache line sharing (true or false) that the code exhibits.
b. Rewrite the code to reduce the number of misses to elements of the array word_count.
c. Identify a manual fix you can make to the code to rid it of any false sharing.

5.29 [15] <5.4> Assume a directory-based cache coherence protocol. The directory currently has information that indicates that processor P1 has the data in "exclusive" mode. If the directory now gets a request for the same cache block from processor P1, what could this mean? What should the directory controller do? (Such cases are called "race conditions" and are the reason why coherence protocols are so hard to design and verify.)

5.30 [20] <5.4> A directory controller can send invalidates for lines that have been replaced by the local cache controller. To avoid such messages, and to keep the directory consistent, replacement hints are used. Such messages tell the controller that a block has been replaced. Modify the directory coherence protocol of Section 5.4 to use such replacement hints.

5.31 [20/15/20/15] <5.4> One downside of a straightforward implementation of directories using fully populated bit vectors is that the total size of the directory information scales as the product: processor count × memory blocks. If memory grows linearly with processor count, the total size of the directory grows quadratically in the processor count. In practice, because the directory needs only 1 bit per memory block (which is typically 32−128 bytes), this problem is not serious for small-to-moderate processor counts. For example, assuming a 128-byte block, and P processors, the amount of directory storage compared to main memory is P/(128*8) = P/1024, which is 12.5% overhead for P = 128 processors. We can avoid this problem by observing that we need to keep only an amount of information that is proportional to the cache size of each processor. We explore some solutions in these exercises.

a. [20] <5.4> One method to obtain a scalable directory protocol is to organize the multiprocessor as a logical hierarchy with the processors as leaves of the hierarchy and directories positioned at the root of each subtree. The directory at each subtree records which descendants cache which memory blocks. It also records the memory blocks—with a home in that subtree—that are cached outside the subtree. Compute the amount of storage needed to record the processor information for the directories, assuming that each directory is fully associative. Your answer should incorporate both the number of nodes at each level of the hierarchy as well as the total number of nodes.
b. [15] <5.4> Another approach to reducing the directory size is to allow only a limited number of the directory's memory blocks to be shared at any given time. Implement the directory as a four-way set-associative cache storing full bit vectors. If a directory cache miss occurs, choose a directory entry and invalidate the entry. Describe how this organization will work, elaborating what will happen as a block is read, written, replaced, and written back to memory. Modify the protocol in Figure 5.20 to reflect the new transitions required by this directory organization.
c. [20] <5.4> Rather than reducing the number of directory entries, we can implement bit vectors that are not dense. For example, we can set every directory entry to 9 bits. If a block is cached in only one node outside its home, this field contains the node number. If the block is cached in more than one node outside its home, this field is a bit vector, with each bit indicating a group of eight processors, at least one of which caches the block. Illustrate how this scheme would work for a 64-processor DSM machine that consists of eight 8-processor groups.
d. [15] An extreme approach to reducing the directory size is to implement an "empty" directory; that is, the directory in every processor does not store any memory states. It receives requests and forwards them as *appropriate.* What is the benefit of having such a directory over having no directory at all for a DSM system?

5.32 [10] <5.5> Implement the classical compare-and-swap instruction using the *load linked/store conditional* instruction pair.

5.33 [15] <5.5> One performance optimization commonly used is to pad synchronization variables so as not to have any other useful data in the same cache line. Construct an example demonstrating that this optimization can be extremely useful in some situations. Assume a snoopy write invalidate protocol.

5.34 [30] <5.5> One possible implementation of the *load linked/store conditional pair* for multicore processors is to constrain these instructions to using uncached memory operations. A monitor unit intercepts all reads and writes from any core to the memory. It keeps track of the source of the *load linked* instructions and whether any intervening stores occur between the *load linked* and its corresponding *store conditional* instruction. The monitor can prevent any failing store conditional from

writing any data and can use the interconnect signals to inform the processor that this store failed.

Design such a monitor for a memory system supporting a four-core SMP. Take into account that, generally, read and write requests can have different data sizes (4/8/16/32 bytes). Any memory location can be the target of a *load linked/store conditional* pair, and the memory monitor should assume that *load linked/store conditional* references to any location can possibly be interleaved with regular accesses to the same location. The monitor complexity should be independent of the memory size.

5.35 [25] <5.5> Prove that, in a two-level cache hierarchy where L1 is closer to the processor, inclusion is maintained with no extra action if L2 has at least as much associativity as L1, both caches use LRU replacement, and both caches have the same block sizes.

[Discussion] <5> When trying to perform detailed performance evaluation of a multiprocessor system, system designers use one of three tools: analytical models, trace-driven simulation, and execution-driven simulation. Analytical models use mathematical expressions to model the behavior of programs. Trace-driven simulations run the applications on a real machine and generate a trace, typically of memory operations. These traces can be replayed through a cache simulator or a simulator with a simple processor model to predict the performance of the system when various parameters are changed. Execution-driven simulators simulate the entire execution, maintaining an equivalent structure for the processor state and so on.

a. What are the accuracy/speed trade-offs between these approaches?
b. CPU traces, if not carefully collected, can exhibit artifacts of the system they are collected on. Discuss this issue while using branch-prediction and spin-wait synchronization as examples. (Hint: The program itself is not available to a pure CPU trace; just the trace is available.)

5.36 [40] <5.7, 5.9> Multiprocessors and clusters usually show performance increases as you increase the number of the processors, with the ideal being n times speedup for n processors. The goal of this biased benchmark is to make a program that gets worse performance as you add processors. For example, this means that one processor on the multiprocessor or cluster runs the program fastest, two are slower, four are slower than two, and so on. What are the key performance characteristics for each organization that give inverse linear speedup?