5

Shared Memory Multiprocessors

The most prevalent form of parallel architecture is the multiprocessor of small to moderate scale that provides a global physical address space and symmetric access to all of main memory from any processor, often called a *symmetric multiprocessor* or SMP. Every processor has its own cache, and all the processors and memory modules attach to the same interconnect, which is usually a shared bus. SMPs dominate the server market and are becoming more common on the desktop. They are also important building blocks for larger-scale systems. The efficient sharing of resources, such as memory and processors, makes these machines attractive as "throughput engines" for multiple sequential jobs with varying memory and CPU requirements. The ability to access all shared data efficiently from any of the processors using ordinary loads and stores, together with the automatic movement and replication of shared data in the local caches, makes them attractive for parallel programming. These features are also very useful for the operating system, whose different processes share data structures and can easily run on different processors.

From the viewpoint of the layers of the communication architecture in Figure 5.1, the shared address space programming model is supported directly by hardware. User processes can read and write shared virtual addresses, and these operations are realized by individual loads and stores of shared physical addresses. In fact, the relationship between the programming model and the hardware operation is so close that they both are often referred to simply as "shared memory." A message-passing programming model can be supported by an intervening software layer—typically a run-time library—that treats large portions of the shared address space as private to each process and manages some portions explicitly as per-process message buffers. A send/receive operation pair is realized by copying data between these buffers. The operating system need not be involved since address translation and protection on the shared buffers is provided by the hardware. For portability, most message-passing programming interfaces have indeed been implemented on popular SMPs. In fact, such implementations often deliver higher message-passing performance than traditional, distributed-memory message-passing systems—as long as contention for the shared bus and memory does not become a bottleneck—largely because of the lack of operating system involvement in communication. The operating system is still used for input/output and multiprogramming support.

Since all communication and local computation generates memory accesses in a shared address space, from a system architect's perspective the key high-level design

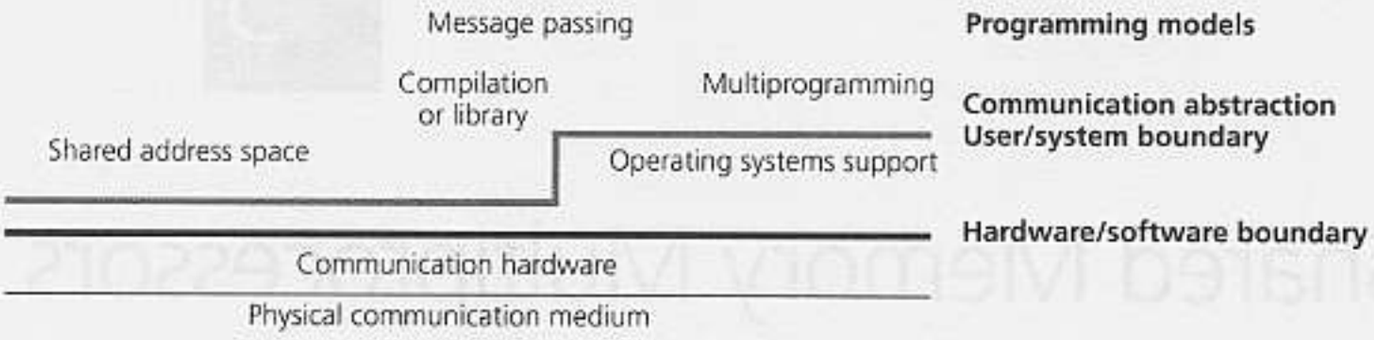

FIGURE 5.1 Layers of abstraction of the communication architecture for bus-based SMPs. A shared address space is supported directly in hardware, while message passing is supported in software.

issue is the organization of the extended memory hierarchy. In general, memory hierarchies in multiprocessors fall primarily into four categories, as shown in Figure 5.2, which correspond loosely to the scale of the multiprocessor being considered. The first three are symmetric multiprocessors (all of main memory is equally far away from all processors), while the fourth is not.

In the shared cache approach (Figure 5.2[a]), the interconnect is located between the processors and a shared first-level cache, which in turn connects to a shared main memory subsystem. Both the cache and the main memory system may be interleaved to increase available bandwidth. This approach has been used for connecting very small numbers of processors (2–8). In the mid-1980s, it was a common technique for connecting a couple of processors on a board; today, it is a possible strategy for a multiprocessor-on-a-chip, where a small number of processors on the same chip share an on-chip first-level cache. However, it applies only at a very small scale, both because the interconnect between the processors and the shared first-level cache is on the critical path that determines the latency of cache access and because the shared cache must deliver tremendous bandwidth to the multiple processors accessing it simultaneously.

In the bus-based shared memory approach (Figure 5.2[b]), the interconnect is a shared bus located between the processor's private caches (or cache hierarchies) and the shared main memory subsystem. This approach has been widely used for small- to medium-scale multiprocessors consisting of up to 20 or 30 processors. It is the dominant form of parallel machine sold today, and considerable design effort has been invested in essentially all modern microprocessors to support "cache-coherent" shared memory configurations. For example, the Intel Pentium Pro processor can attach to a coherent shared bus without any glue logic, and low-cost bus-based machines that use these processors have greatly increased the popularity of this approach. The scaling limit for these machines comes primarily due to bandwidth limitations of the shared bus and memory system.

The last two approaches are intended to be scalable to many processing nodes. The dancehall approach also places the interconnect between the caches and main memory, but the interconnect is now a scalable point-to-point network rather than a bus, and memory is divided into many logical modules that connect to logically dif-

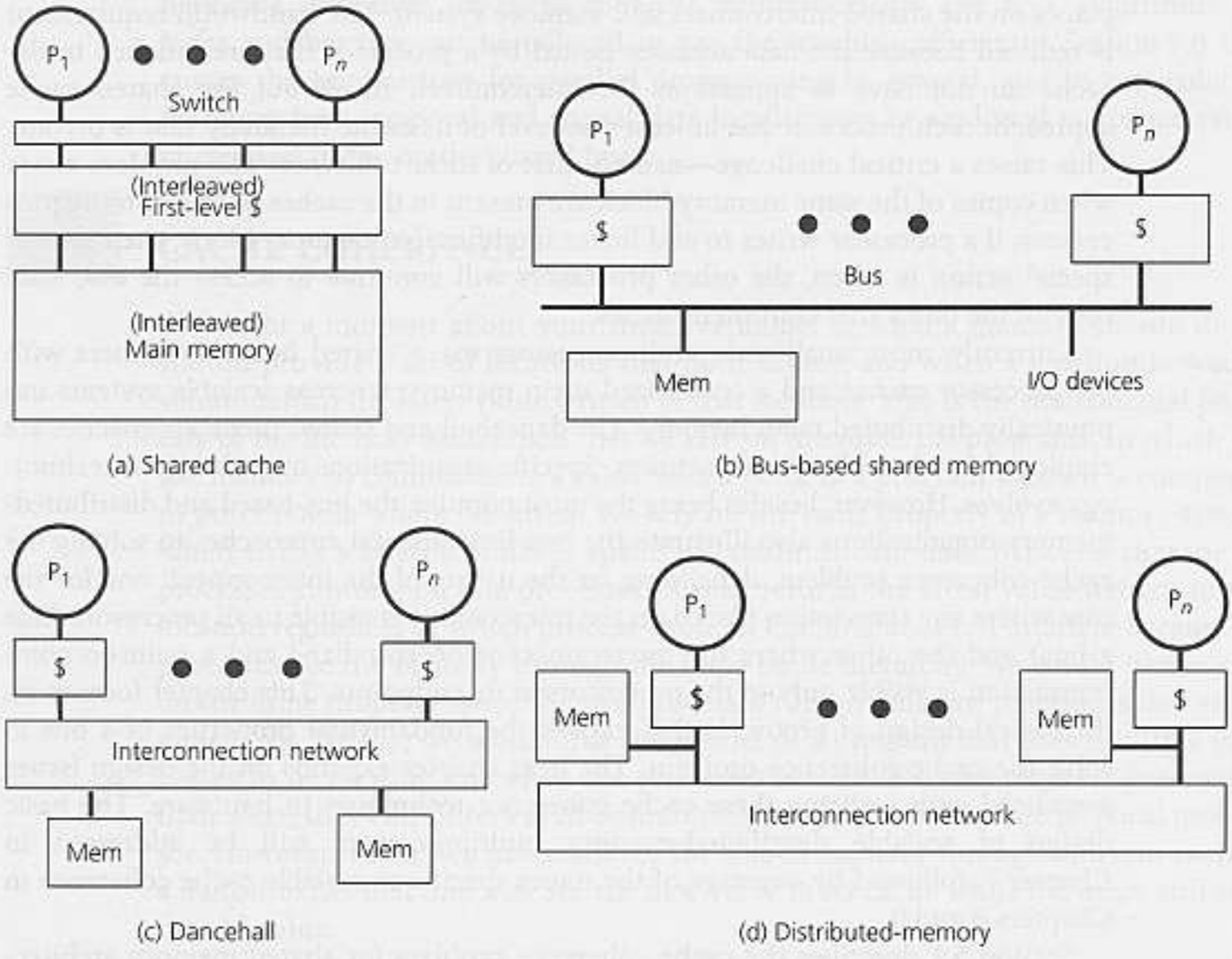

FIGURE 5.2 Common extended memory hierarchies found in multiprocessors

ferent points in the interconnect (Figure 5.2[c]). This approach is symmetric—all of main memory is uniformly far away from all processors—but its limitation is that all of memory is indeed *far* away from all processors. Especially in large systems, several "hops" or switches in the interconnect must be traversed to reach any memory module from any processor. The fourth approach, distributed-memory, is not symmetric. A scalable interconnect is located between processing nodes, but each node has its own local portion of the global main memory to which it has faster access (Figure 5.2[d]). By exploiting locality in the distribution of data, most cache misses may be satisfied in the local memory and may not have to traverse the network. This design is most attractive for scalable multiprocessors, and several chapters are devoted to the topic later in the book. Of course, it is also possible to combine multiple approaches into a single machine design—for example, a distributed-memory machine whose individual nodes are bus-based SMPs or a machine in which processors share a cache at a level of the hierarchy other than the first level.

In all cases, caches play an essential role in reducing the average data access time as seen by the processor and in reducing the bandwidth requirement each processor

places on the shared interconnect and memory system. The bandwidth requirement is reduced because the data accesses issued by a processor that are satisfied in the cache do not have to appear on the interconnect. In all but the shared cache approach, each processor has at least one level of its cache hierarchy that is private. This raises a critical challenge—namely, that of *cache coherence*. The problem arises when copies of the same memory block are present in the caches of one or more processors; if a processor writes to and hence modifies that memory block, then, unless special action is taken, the other processors will continue to access the old, stale copy of the block that is in their caches.

Currently, most small-scale multiprocessors use a shared bus interconnect with per-processor caches and a centralized main memory, whereas scalable systems use physically distributed main memory. The dancehall and shared cache approaches are employed in relatively specific settings. Specific organizations may change as technology evolves. However, besides being the most popular, the bus-based and distributed-memory organizations also illustrate the two fundamental approaches to solving the cache coherence problem, depending on the nature of the interconnect: one for the case where any transaction placed on the interconnect is visible to all processors (like a bus) and the other where the interconnect is decentralized and a point-to-point transaction is visible only to the processors at its endpoints. This chapter focuses on the logical design of protocols that exploit the fundamental properties of a bus to solve the cache coherence problem. The next chapter expands on the design issues associated with realizing these cache coherence techniques in hardware. The basic design of scalable distributed-memory multiprocessors will be addressed in Chapter 7, followed by coverage of the issues specific to scalable cache coherence in Chapters 8 and 9.

Section 5.1 describes the cache coherence problem for shared memory architectures in detail and describes the simplest example of what are called *snooping* cache coherence protocols. Coherence is not only a key hardware design concept but is a necessary part of our intuitive notion of the abstraction of memory. However, parallel software often makes stronger assumptions than coherence about how memory behaves. Section 5.2 extends the discussion of ordering begun in Chapter 1 and introduces the concept of memory consistency, which defines the semantics of shared address space. This issue has become increasingly important in computer architecture and compiler design; a large fraction of the reference manuals for most recent instruction set architectures is devoted to the memory consistency model. Once the abstractions and concepts are defined, Section 5.3 presents the design space for more realistic snooping protocols and shows how they satisfy the conditions for coherence as well as for a useful consistency model. It describes the operation of commonly used protocols at the logical state transition level. The techniques used for the quantitative evaluation of several design trade-offs at this level are illustrated in Section 5.4, using aspects of the methodology for workload-driven evaluation from Chapter 4.

The latter portions of the chapter examine the implications that cache-coherent shared memory architectures have for the software that runs on them. Section 5.5 examines how the low-level synchronization operations make use of the available

hardware primitives on cache-coherent multiprocessors and how algorithms for locks and barriers can be tailored to use the machine efficiently. Section 5.6 discusses the implications for parallel programming in general, and in particular, it discusses how temporal and spatial data locality may be exploited to reduce cache misses and traffic on the shared bus.

5.1 CACHE COHERENCE

Think for a moment about your intuitive model of what a memory should do. It should provide a set of locations that hold values, and when a location is read it should return the latest value written to that location. This is the fundamental property of the memory abstraction that we rely on in sequential programs, in which we use memory to communicate a value from a point in a program where it is computed to other points where it is used. We rely on the same property of a memory system when using a shared address space to communicate data between threads or processes running on one processor. A read returns the latest value written to the location regardless of which process wrote it. Caching does not interfere because all processes see the memory through the same cache hierarchy. We would like to rely on the same property when the two processes run on different processors that share a memory. That is, we would like the results of a program that uses multiple processes to be no different when the processes run on different physical processors than when they run (interleaved or multiprogrammed) on the same physical processor. However, when two processes see the shared memory through different caches, a danger exists that one may see the new value in its cache while the other still sees the old value.

5.1.1 The Cache Coherence Problem

The cache coherence problem in multiprocessors is both pervasive and performance critical. It is illustrated in Example 5.1.

EXAMPLE 5.1 Figure 5.3 shows three processors with caches connected via a bus to shared main memory. A sequence of accesses to location *u* is made by the processors. First, processor P_1 reads *u* from main memory, bringing a copy into its cache. Then processor P_3 reads *u* from main memory, bringing a copy into its cache. Then processor P_3 writes location *u*, changing its value from 5 to 7. With a write-through cache, this will cause the main memory location to be updated; however, when processor P_1 reads location *u* again (action 4), it will unfortunately read the stale value 5 from its own cache instead of the correct value 7 from main memory. This is a cache coherence problem. What happens if the caches are write back instead of write through?

Answer The situation is even worse with write-back caches. P_3's write would merely set the dirty (or modified) bit associated with the cache block holding location *u* and would not update main memory right away. Only when this cache block is subsequently replaced from P_3's cache would its contents be written back to main memory. Thus, not only will P_1 read the stale value, but when processor P_2 reads

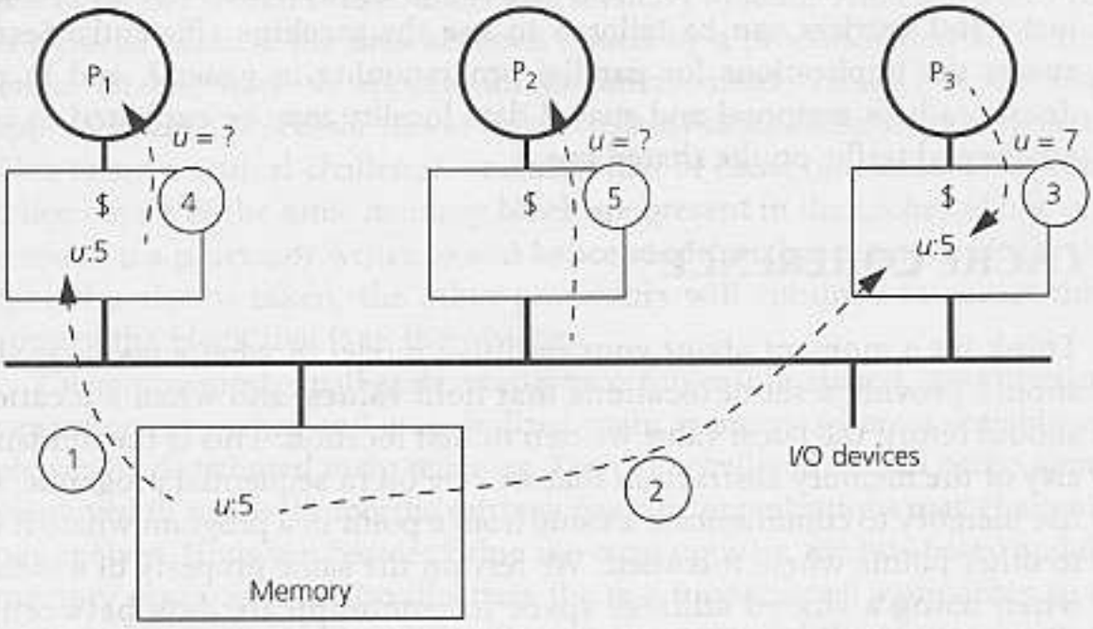

FIGURE 5.3 Example cache coherence problem. The figure shows three processors with caches connected by a bus to main memory. *u* is a location in memory whose contents are being read and written by the processors. The sequence in which reads and writes are done is indicated by the number listed inside the circles placed next to the arc. It is easy to see that unless special action is taken when P_3 updates the value of *u* to 7, P_1 will subsequently continue to read the stale value out of its cache, and P_2 will also read a stale value out of main memory.

location *u* (action 5), it will miss in its cache and read the stale value of 5 from main memory instead of 7. Finally, if multiple processors write distinct values to location *u* in their write-back caches, the final value that will reach main memory will be determined by the order in which the cache blocks containing *u* are replaced and will have nothing to do with the order in which the writes to *u* occur. ■

Clearly, the behavior described in Example 5.1 violates our intuitive notion of what a memory should do. In fact, cache coherence problems arise even in uniprocessors when I/O operations occur. Most I/O transfers are performed by direct memory access (DMA) devices that move data between memory and the peripheral component without involving the processor. When the DMA device writes to a location in main memory, unless special action is taken, the processor may continue to see the old value if that location was previously present in its cache. With write-back caches, a DMA device may read a stale value for a location from main memory because the latest value for that location is in the processor's cache. Since I/O operations are much less frequent than memory operations, several coarse solutions have been adopted in uniprocessors. For example, segments of memory space used for I/O may be marked as "uncacheable" (i.e., they do not enter the processor cache), or the processor may always use uncached load and store operations for locations used to communicate with I/O devices. For I/O devices that transfer large blocks of data at a time, such as disks, operating system support is often enlisted to ensure coherence. In many systems, the pages of memory from/to which the data is

to be transferred are flushed by the operating system from the processor's cache before the I/O is allowed to proceed. In still other systems, all I/O traffic is made to flow through the processor cache hierarchy, thus maintaining coherence. This, of course, pollutes the cache hierarchy with data that may not be of immediate interest to the processor. Fortunately, the techniques and support used to solve the multiprocessor cache coherence problem also solve the I/O coherence problem. Essentially all microprocessors today provide support for multiprocessor cache coherence.

In multiprocessors, reading and writing of shared variables by different processors is expected to be a frequent event since it is the way that multiple processes belonging to a parallel application communicate with each other. Therefore, we do not want to disallow caching of shared data or to invoke the operating system on all shared references. Rather, cache coherence needs to be addressed as a basic hardware design issue; for example, stale cached copies of a shared location (like the copy of u in P_1's cache in Example 5.1) must be eliminated when the location is modified, either by invalidating them or updating them with the new value. In fact, the operating system itself benefits greatly from transparent, hardware-supported coherence of its data structures.

Before we explore techniques to provide coherence, it is useful to define the coherence property more precisely. Our intuitive notion that "each read should return the last value written to that location" is problematic for parallel architecture because "last" may not be well defined. Two different processors might write to the same location at the same instant, or one processor may read so soon after another writes that, due to the speed of light and other factors, there isn't time to propagate the invalidation or update to the reader. Even in the sequential case, "last" is not a chronological or physical notion but refers to latest in program order. For now, we can think of program order within a process as the order in which memory operations occur in the machine language program. The subtleties of program order are elaborated further in Section 5.2. The challenge in the parallel case is that, while program order is defined for the operations within each individual process, in order to define the semantics of a coherent memory system we need to make sense of the collection of program orders.

Let us first review the definitions of some terms in the context of uniprocessor memory systems so that we can extend the definitions for multiprocessors. By *memory operation*, we mean a single read (load), write (store), or read-modify-write access to a memory location. Instructions that perform multiple reads and writes, such as those that appear in many complex instruction sets, can be viewed as broken down into multiple memory operations, and the order in which these memory operations are executed is specified by the instruction. These memory operations within an instruction are assumed to execute atomically with respect to each other in the specified order; that is, all aspects of one appear to execute before any aspect of the next. A memory operation *issues* when it leaves the processor's internal environment and is presented to the memory system, which includes the caches, write buffers, bus, and memory modules. A very important point for ordering is that the only way the processor observes the state of the memory system is by issuing memory operations (e.g., reads); thus, for a memory operation to be *performed with respect to the*

processor means that it appears to have taken place, as far as the processor can tell from the memory operations it issues. In particular, a write operation is said to perform with respect to the processor when a subsequent read by the processor returns the value produced by either that write or a later write. A read operation is said to perform with respect to the processor when subsequent writes issued by the processor cannot affect the value returned by the read. Notice that in neither case do we specify that the physical location in the memory chip has been accessed or that specific bits of hardware have changed their values. Also, "subsequent" is well defined in the sequential case since reads and writes are ordered by the program order.

The same definitions for memory operations issuing and performing with respect to a processor apply in the parallel case; we can simply replace "the processor" with "a processor" in the definitions. The problem is that "subsequent" and "last" are not yet well defined since we do not have one program order; rather, we have separate program orders for every process, and these program orders interact when accessing the memory system. One way to sharpen our idea of a coherent memory system is to picture what would happen if there were a single shared memory and no caches. Every write and every read to a memory location would access the physical location at main memory. The operation would be performed with respect to all processors at this point and would therefore be said to *complete*. Thus, the memory would impose a serial order on all the read and write operations from all processors to the location. Moreover, the reads and writes to the location from any individual processor should be in program order within this overall serial order. In this case, then, the main memory location provides a natural point in the hardware to determine the order across processes of operations to that location. We have no reason to believe that the memory system should interleave accesses from different processors in a particular way, so any interleaving that preserves the individual program orders is reasonable. We do assume some basic fairness; eventually, the operations from each processor should be performed. Our intuitive notion of "last" can be viewed as most recent in a hypothetical serial order that maintains these properties, and "subsequent" can be defined similarly. Since this serial order must be consistent, it is important that all processors see the writes to a location in the same order (if they bother to look, i.e., to read the location).

The appearance of such a total, serial order on operations to a location is what we expect from any coherent memory system. Of course, the total order need not actually be constructed at any given point in the machine while executing the program. Particularly in a system with caches, we do not want main memory to see all the memory operations, and we want to avoid serialization whenever possible. We just need to make sure that the program behaves as if some serial order was enforced.

More formally, we say that a multiprocessor memory system is *coherent* if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processes into a total order) that is consistent with the results of the execution and in which

1. operations issued by any particular process occur in the order in which they were issued to the memory system by that process, and

2. the value returned by each read operation is the value written by the last write to that location in the serial order.

Two properties are implicit in the definition of coherence: *write propagation* means that writes become visible to other processes; *write serialization* means that all writes to a location (from the same or different processes) are seen in the same order by all processes. For example, write serialization means that if read operations by process P_1 to a location see the value produced by write $w1$ (from P_2, say) before the value produced by write $w2$ (from P_3, say), then reads by another process P_4 (or P_2 or P_3) also should not be able to see $w2$ before $w1$. There is no need for an analogous concept of read serialization since the effects of reads are not visible to any process but the one issuing the read.

The *results* of a program can be viewed as the values returned by the read operations in it, perhaps augmented with an implicit set of reads to all locations at the end of the program. From the results, we cannot determine the order in which operations were actually executed by the machine or exactly when bits changed, only the order in which they appear to execute. Fortunately, this is all that matters since this is all that processors can detect. This concept will become even more important when we discuss memory consistency models.

5.1.2 Cache Coherence through Bus Snooping

Having defined the memory coherence property, let us examine techniques to solve the cache coherence problem. For instance, in Figure 5.3, how do we ensure that P_1 and P_2 see the value that P_3 wrote? In fact, a simple and elegant solution to cache coherence arises from the very nature of a bus. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction, for example, every read or write on the shared bus. When a processor issues a request to its cache, the cache controller examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having all cache controllers "snoop" on the bus and monitor the transactions, as illustrated in Figure 5.4 (Goodman 1983). A snooping cache controller may take action if a bus transaction is *relevant* to it—that is, if it involves a memory block of which it has a copy in its cache. Thus, P_1 may take an action, such as invalidating or updating its copy of the location, if it sees the write from P_3. In fact, since the allocation and replacement of data in caches is managed at the granularity of a cache block (usually several words long) and cache misses fetch a block of data, most often coherence is maintained at the granularity of a cache block as well. In other words, either an entire cache block is in valid state in the cache or none of it is. Thus, a cache block is the granularity of allocation in the cache, of data transfer between caches, and of coherence.

The key properties of a bus that support coherence are the following. First, all transactions that appear on the bus are visible to all cache controllers. Second, they are visible to all controllers in the same order (the order in which they appear on the bus). A coherence protocol must guarantee that all the "necessary" transactions in

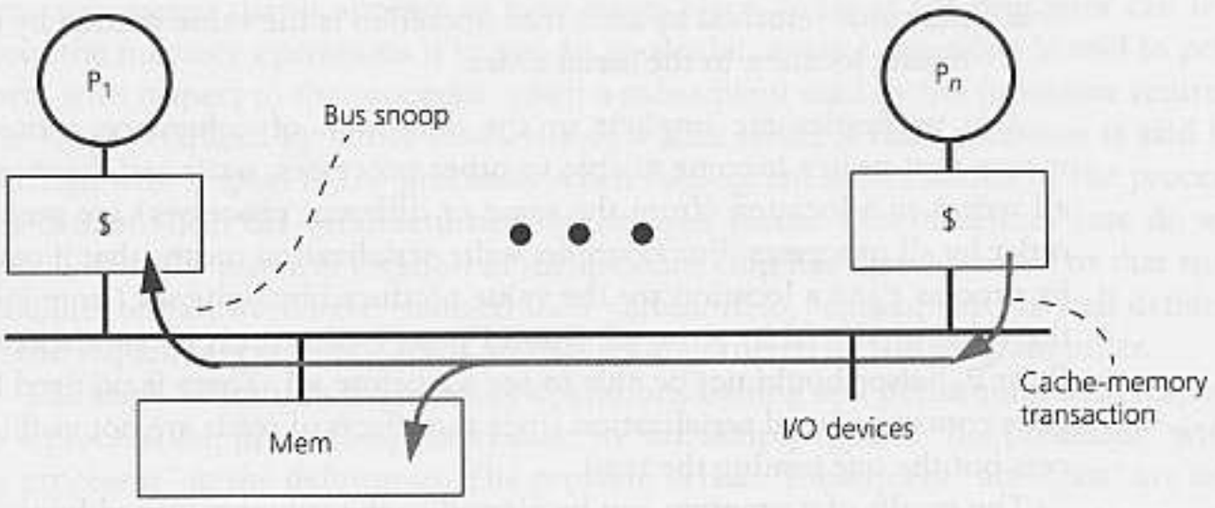

FIGURE 5.4 A snooping cache-coherent multiprocessor. Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously "snoops" on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent. The gray arrows show the transaction being placed on the bus and accepted by main memory, as in a uniprocessor system. The black arrow indicates the snoop.

fact appear on the bus, in response to memory operations, and that the controllers take the appropriate actions when they see a relevant transaction.

The simplest illustration of maintaining coherence is a system that has single-level write-through caches. It is basically the approach followed by the first commercial bus-based SMPs in the mid-1980s. In this case, every write operation causes a write transaction to appear on the bus, so every cache controller observes every write (thus providing write propagation). If a snooping cache has a copy of the block, it either invalidates or updates its copy. Protocols that invalidate cached copies (other than the writer's copy) on a write are called *invalidation-based protocols*, whereas those that update other cached copies are called *update-based protocols*. In either case, the next time the processor with the copy accesses the block, it will see the most recent value, either through a miss or because the updated value is in its cache. Main memory always has valid data, so the cache need not take any action when it observes a read on the bus. Example 5.2 illustrates how the coherence problem in Figure 5.3 is solved with write-through caches.

EXAMPLE 5.2 Consider the scenario presented in Figure 5.3. Assuming write-through caches, show how the bus may be used to provide coherence using an invalidation-based protocol.

Answer When processor P_3 writes 7 to location u, P_3's cache controller generates a bus transaction to update memory. Observing this bus transaction as relevant and as a write transaction, P_1's cache controller invalidates its own copy of the block containing u. The main memory controller will update the value it has stored for location u to 7. Subsequent reads to u from processors P_1 and P_2 (actions 4 and 5) will both miss in their private caches and get the correct value of 7 from the main memory. ■

The check to determine if a bus transaction is relevant to a cache is essentially the same tag match that is performed for a request from the processor. The action taken may involve invalidating or updating the contents or state of that cache block and/or supplying the latest value for that block from the cache to the bus.

A snoopy cache coherence protocol ties together two basic facets of computer architecture that are also found in uniprocessors: bus transactions and the state transition diagram associated with a cache block. Recall that the first component—the bus transaction—consists of three phases: arbitration, command/address, and data. In the arbitration phase, devices that desire to initiate a transaction assert their bus request, and the bus arbiter selects one of these and responds by asserting its grant signal. Upon grant, the selected device places the command, for example, read or write, and the associated address on the bus command and address lines. All devices observe the address and, in a uniprocessor, one of them recognizes that it is responsible for the particular address. For a read transaction, the address phase is followed by data transfer. Write transactions vary from bus to bus according to whether the data is transferred during or after the address phase. For most buses, a responding device can assert a wait signal to hold off the data transfer until it is ready. This wait signal is different from the other bus signals because it is a wired-OR across all the processors; that is, it is a logical 1 if any device asserts it. The initiator does not need to know which responding device is participating in the transfer, only that there is one and whether it is ready.

The second basic facet of computer architecture leveraged by a cache coherence protocol is that each block in a uniprocessor cache has a state associated with it, along with the tag and data, which indicates the disposition of the block, (e.g., invalid, valid, dirty). The cache policy is defined by the *cache block state transition diagram*, which is a finite state machine specifying how the disposition of a block changes. Transitions for a cache block occur upon access to that block or to an address that maps to the same cache line as that block. (We refer to a cache block as the actual data, and a line as the fixed storage in the hardware cache, in exact analogy with a page and a page frame in main memory.) While only blocks that are actually in cache lines have hardware state information, logically, all blocks that are not resident in the cache can be viewed as being in either a special "not present" state or in the "invalid" state. In a uniprocessor system, for a write-through, write-no-allocate cache (Hennessy and Patterson 1996), only two states are required: valid and invalid. Initially, all the blocks are invalid. When a processor read operation misses, a bus transaction is generated to load the block from memory and the block is marked valid. Writes generate a bus transaction to update memory, and they also update the cache block if it is present in the valid state. Writes do not change the state of the block. If a block is replaced, it may be marked invalid until the memory provides the new block, whereupon it becomes valid. A write-back cache requires an additional state per cache line, indicating a "dirty" or modified block.

In a multiprocessor system, a block has a state in each cache, and these cache states change according to the state transition diagram. Thus, we can think of a block's cache state as being a vector of p states instead of a single state, where p is the number of caches. The cache state is manipulated by a set of p distributed finite state

machines, implemented by the cache controllers. The state machine or state transition diagram that governs the state changes is the same for all blocks and all caches, but the current state of a block in different caches is different. As before, if a block is not present in a cache we can assume it to be in a special "not present" state or even in the invalid state.

In a snooping cache coherence scheme, each cache controller receives two sets of inputs: the processor issues memory requests, and the bus snooper informs about bus transactions from other caches. In response to either, the controller may update the state of the appropriate block in the cache according to the current state and the state transition diagram. It may also take an action. For example, it responds to the processor with the requested data, potentially generating new bus transactions to obtain the data. It responds to bus transactions by updating its state and sometimes intervenes in completing the transaction. Thus, a *snooping protocol* is a distributed algorithm represented by a collection of cooperating finite state machines. It is specified by the following components:

- the set of states associated with memory blocks in the local caches
- the state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction and produces as output the next state for the cache block
- the actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, the cache, and the processor design

The different state machines for a block are coordinated by bus transactions.

A simple invalidation-based protocol for a coherent write-through, write-no-allocate cache is described by the state transition diagram in Figure 5.5. As in the uniprocessor case, each cache block has only two states: invalid (I) and valid (V) (the "not present" state is assumed to be the same as invalid). The transitions are marked with the input that causes the transition and the output that is generated with the transition. For example, when a controller sees a read from its processor miss in the cache, a BusRd transaction is generated, and upon completion of this transaction the block transitions up to the valid state. Whenever the controller sees a processor write to a location, a bus transaction is generated that updates that location in main memory with no change of state. The key enhancement to the uniprocessor state diagram is that when the bus snooper sees a write transaction on the bus for a memory block that is cached locally, the controller sets the cache state for that block to invalid, thereby effectively discarding its copy. (Figure 5.5 shows this bus-induced transition with a dashed arc.) By extension, if any processor generates a write for a block that is cached by any of the others, all of the others will invalidate their copies. Thus, multiple simultaneous readers of a block may coexist without generating bus transactions or invalidations, but a write will eliminate all other cached copies.

To see how this simple write-through invalidation protocol provides coherence, we need to show that for any execution under the protocol a total order on the mem-

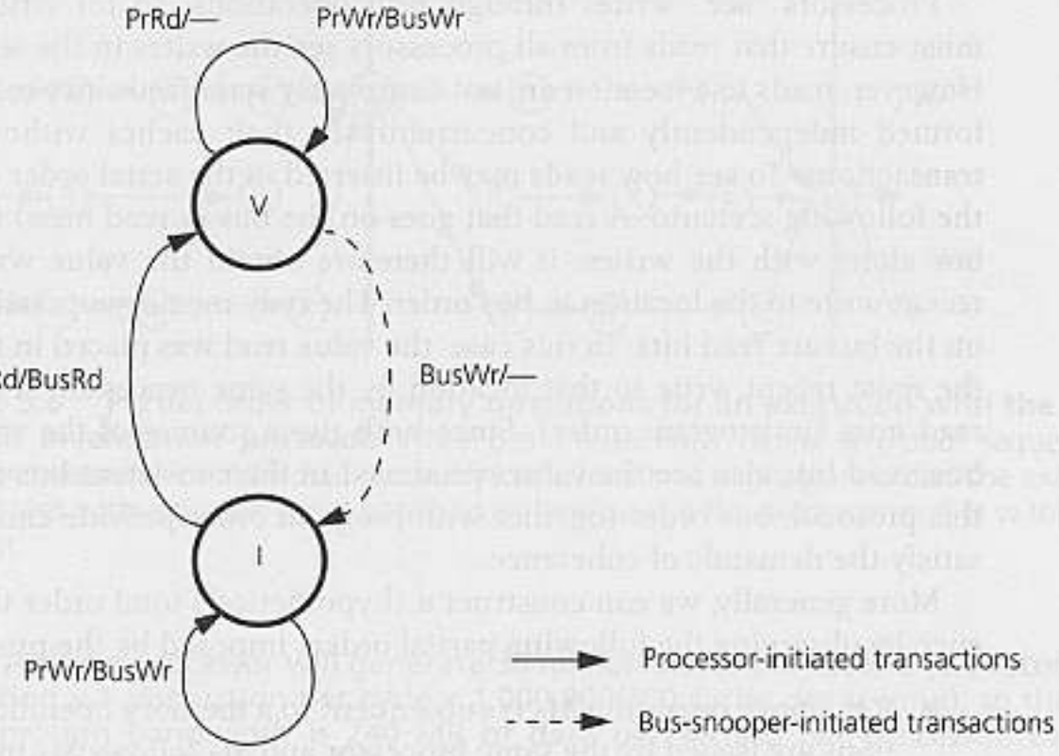

FIGURE 5.5 Snoopy coherence for a multiprocessor with write-through, write-no-allocate caches. There are two states, valid (V) and invalid (I), with intuitive semantics. The notation *A/B* (e.g., PrRd/BusRd) means if *A* is observed, then transaction *B* is generated. From the processor side, the requests can be read (PrRd) or write (PrWr). From the bus side, the cache controller may observe/generate transactions bus read (BusRd) or bus write (BusWr).

ory operations for a location can be constructed that satisfies the program order and write serialization conditions. Let us assume for the present discussion that both bus transactions and the memory operations are atomic. That is, only one transaction is in progress on the bus at a time: once a request is placed on the bus, all phases of the transaction, including the data response, complete before any other request from any processor is allowed access to the bus (such a bus with atomic transactions is called an *atomic bus*). Also, a processor waits until its previous memory operation is complete before issuing another memory operation. With single-level caches, it is also natural to assume that invalidations are applied to the caches, and hence the write completes during the bus transaction itself. (These assumptions will be continued throughout this chapter and will be relaxed when we look at protocol implementations in more detail and study high-performance designs with greater concurrency in Chapter 6.) Finally, we may assume that the memory handles writes and reads in the order in which they are presented by the bus.

In the write-through protocol, all writes appear on the bus. Since only one bus transaction is in progress at a time, in any execution all writes to a location are serialized (consistently) by the order in which they appear on the shared bus, called the *bus order*. Since each snooping cache controller performs the invalidation during the bus transaction, invalidations are performed by all cache controllers in bus order.

Processors "see" writes through read operations, so for write serialization we must ensure that reads from all processors see the writes in the serialized bus order. However, reads to a location are not completely serialized since read hits may be performed independently and concurrently in their caches without generating bus transactions. To see how reads may be inserted in the serial order of writes, consider the following scenario. A read that goes on the bus (a read miss) is serialized by the bus along with the writes; it will therefore obtain the value written by the most recent write to the location in bus order. The only memory operations that do not go on the bus are read hits. In this case, the value read was placed in the cache by either the most recent write to that location by the same processor or by its most recent read miss (in program order). Since both these sources of the value appear on the bus, read hits also see the values produced in the consistent bus order. Thus, under this protocol, bus order together with program order provide enough constraints to satisfy the demands of coherence.

More generally, we can construct a (hypothetical) total order that satisfies coherence by observing the following partial orders imposed by the protocol:

- A memory operation M_2 is subsequent to a memory operation M_1 if the operations are issued by the same processor and M_2 follows M_1 in program order.
- A read operation is subsequent to a write operation W if the read generates a bus transaction that follows that for W.
- A write operation is subsequent to a read or write operation M if M generates a bus transaction and the bus transaction for the write follows that for M.
- A write operation is subsequent to a read operation if the read does not generate a bus transaction (is a hit) and is not already separated from the write by another bus transaction.

Any serial order that preserves the resulting partial order is coherent. The "subsequent" ordering relationship is transitive. An illustration of the resulting partial order is depicted in Figure 5.6, where the bus transactions associated with writes segment the individual program orders. The partial order does not constrain the ordering of read bus transactions from different processors that occur between two write transactions, though the bus will likely establish a particular order. In fact, any interleaving of read operations in the segment between two writes is a valid serial order, as long as it obeys program order.

Of course, the problem with this simple write-through approach is that every store instruction goes to memory, which is why most modern microprocessors use write-back caches (at least at the level closest to the bus). This problem is exacerbated in the multiprocessor setting, since every store from every processor consumes precious bandwidth on the shared bus, resulting in poor scalability, as illustrated by Example 5.3.

EXAMPLE 5.3 Consider a superscalar RISC processor issuing two instructions per cycle running at 200 MHz. Suppose the average CPI (clocks per instruction) for this processor is 1, 15% of all instructions are stores, and each store writes 8 bytes of data. How many processors will a 1-GB/s bus be able to support without becoming saturated?

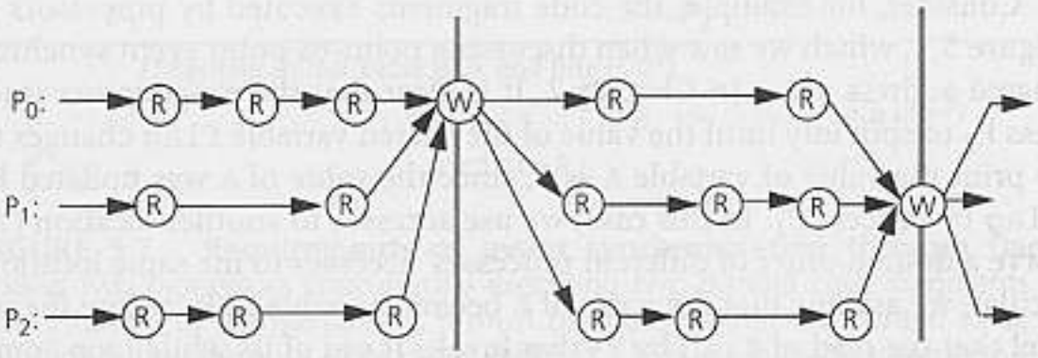

FIGURE 5.6 Partial order of memory operations for an execution with the write-through invalidation protocol. Write bus transactions define a global sequence of events between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving the processor orders within each segment.

Answer A single processor will generate 30 million stores per second (0.15 stores per instruction × 1 instruction per cycle × 1,000,000/200 cycles per second), so the total write-through bandwidth is 240 MB of data per second per processor. Even ignoring address and other information and ignoring read misses, a 1-GB/s bus will therefore support only about four processors. ■

For most applications, a write-back cache would absorb the vast majority of the writes. However, if writes do not go to memory, they do not generate bus transactions, and it is no longer clear how the other caches will observe these modifications and ensure write propagation. Also, when writes to different caches are allowed to occur concurrently, no obvious ordering mechanism exists to sequence the writes. We will need somewhat more sophisticated cache coherence protocols to make the "critical" events visible to the other caches and to ensure write serialization.

The space of protocols for write-back caches is quite large. Before we examine it, let us step back to the more general ordering issue alluded to in the introduction to this chapter and examine the semantics of a shared address space as determined by the memory consistency model.

5.2 MEMORY CONSISTENCY

Coherence, on which we have focused so far, is essential if information is to be transferred between processors by one writing to a location that the other reads. Eventually, the value written will become visible to the reader—indeed to all readers. However, coherence says nothing about when the write will become visible. Often in writing a parallel program, we want to ensure that a read returns the value of a particular write; that is, we want to establish an order between a write and a read. Typically, we use some form of event synchronization to convey this dependence, and we use more than one memory location.

Consider, for example, the code fragments executed by processors P_1 and P_2 in Figure 5.7, which we saw when discussing point-to-point event synchronization in a shared address space in Chapter 2. It is clear that the programmer intends for process P_2 to spin idly until the value of the shared variable `flag` changes to 1 and then to print the value of variable `A` as 1, since the value of `A` was updated before that of `flag` by process P_1. In this case, we use accesses to another location (`flag`) to preserve a desired order of different processes' accesses to the same location (`A`). In particular, we assume that the write of `A` becomes visible to P_2 before the write to `flag` and that the read of `flag` by P_2 that breaks it out of its while loop completes before its read of `A` (a print operation is essentially a read). These program orders within P_1 and P_2's accesses to different locations are not implied by coherence, which, for example, only requires that the new value for `A` eventually become visible to process P_2, not necessarily before the new value of `flag` is observed.

The programmer might try to avoid this issue by using a barrier or other explicit event synchronization, as shown in Figure 5.8. We expect the value of `A` to be printed as 1 since `A` was set to 1 before the barrier. Even this approach has two potential problems, however. First, we are adding assumptions to the meaning of the barrier: not only do processes wait at the barrier until all of them have arrived, they also wait until all writes issued prior to the barrier have become visible to the other processors. Second, a barrier is often built using reads and writes to ordinary shared variables (e.g., `b1` in the figure) rather than with specialized hardware support. In this case, as far as the machine is concerned, it sees only accesses to different shared variables in the compiled code, not a special barrier operation. Coherence does not say anything at all about the order among these accesses.

Clearly, we expect more from a memory system than to "return the last value written" for each location. To establish order among accesses to the same location (say, `A`) by different processes, we sometimes expect a memory system to respect the order of reads and writes to different locations (`A` and `flag` or `A` and `b1`) issued by the same process. Coherence says nothing about the order in which writes to different locations become visible. Similarly, it says nothing about the order in which the reads issued to different locations by P_2 are performed with respect to P_1. Thus, coherence does not in itself prevent an answer of 0 from being printed by either example, which is certainly not what the programmer had in mind.

In other situations, the programmer's intention may not be so clear. Consider the example in Figure 5.9. The accesses made by process P_1 are ordinary writes, and `A` and `B` are not used as flags or synchronization variables. Should we intuitively expect that if the value printed for `B` is 2, then the value printed for `A` is 1? Whatever the answer, the two print statements read different locations and coherence says nothing about the order in which the writes by P_1 become visible to P_2. This example is in fact a fragment from Dekker's algorithm (Tanenbaum and Woodhull 1997) to determine which of two processes arrives first at a critical point as a step in ensuring mutual exclusion. The algorithm relies on writes to distinct locations by a process becoming visible to other processes in the order in which they appear in the

P_1	P_2
*/*Assume initial value of A and flag is 0*/*	
`A = 1;`	`while (flag == 0); /*spin idly*/`
`flag = 1;`	`print A;`

FIGURE 5.7 Requirements of event synchronization through flags. The figure shows two processors concurrently executing two distinct code fragments. For programmer intuition to be maintained, it must be the case that the printed value of `A` is 1. The intuition is that because of program order, if `flag =1` is visible to process P_2, then it must also be the case that `A = 1` is visible to P_2.

P_1	P_2
*/*Assume initial value of A is 0*/*	
`A = 1;`	`. . .`
`- - - BARRIER(b1)- - - - - -`	`BARRIER(b1)- - - - - - - -`
	`print A;`

FIGURE 5.8 Maintaining order among accesses to a location using explicit synchronization through barriers. As in Figure 5.7, the programmer expects the value printed for `A` to be 1 since passing the barrier should imply that the write of `A` by P_1 has already completed and is therefore visible to P_2.

P_1	P_2
*/*Assume initial values of A and B are 0*/*	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

FIGURE 5.9 Order among accesses without synchronization. Here it is less clear what a programmer should expect since neither a flag nor any other explicit event synchronization is used.

program. Clearly, we need something more than coherence to give a shared address space a clear semantics, that is, an ordering model that programmers can use to reason about the possible results and hence the correctness of their programs.

A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e., to become visible to the processors) with respect to one another. This includes operations to the same locations or to different locations and by the same process or different processes, so in this sense memory consistency subsumes coherence.

5.2.1 Sequential Consistency

In the discussion in Chapter 1 of fundamental design issues for a communication architecture, Section 1.4 described informally a desirable ordering model for a shared address space: the reasoning that allows a multithreaded program to work under any possible interleaving on a uniprocessor should hold when some of the threads run in parallel on different processors. The ordering of data accesses within a process was therefore the program order, and that across processes was some interleaving of the program orders. That is, the multiprocessor case should not be able to cause values to become visible to processes in the shared address space in a manner that no sequential interleaving of accesses from different processes can generate. This intuitive model was formalized by Lamport as *sequential consistency* (SC), which is defined as follows (Lamport 1979):[1]

> A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Figure 5.10 depicts the abstraction of memory provided to programmers by a sequentially consistent system (Adve and Gharachorloo 1996). It is similar to the machine model we used to introduce coherence, though now it applies to multiple memory locations. Multiple processes appear to share a single logical memory, even though in the real machine main memory may be distributed across multiple processors, each with their own private caches and buffers. Every process appears to issue and complete memory operations one at a time and atomically in program order; that is, a memory operation does not appear to be issued until the previous one from that process has completed. In addition, the common memory appears to service these requests one at a time in an interleaved manner according to an arbitrary (but hopefully fair) schedule. Memory operations appear *atomic* in this interleaved order; that is, it should appear globally (to all processes) as if one operation in the consistent interleaved order executes and completes before the next one begins.

As with coherence, it is not important in what order memory operations actually issue or even complete. What matters for sequential consistency is that they appear to complete in a manner that satisfies the constraints just described. In the example in Figure 5.9, under SC the result (0,2) for (A,B) would not be allowed—preserving our intuition—since it would then appear that the writes of A and B by process P_1 executed out of program order. However, the memory operations may actually execute and complete in the order 1b, 1a, 2b, 2a. It does not matter that they actually complete out of program order since the results of the execution (1,2) are the same as if the operations were executed and completed in program order. On the other hand, the actual execution order 1b, 2a, 2b, 1a would not be sequentially consistent since it would produce the result (0,2), which is not allowed under SC. Other examples illustrating the intuitiveness of sequential consistency can be found

1. Two closely related concepts in software systems are serializability (Papadimitriou 1979) for concurrent updates to a database and linearizability (Herlihy and Wing 1987) for concurrent objects.

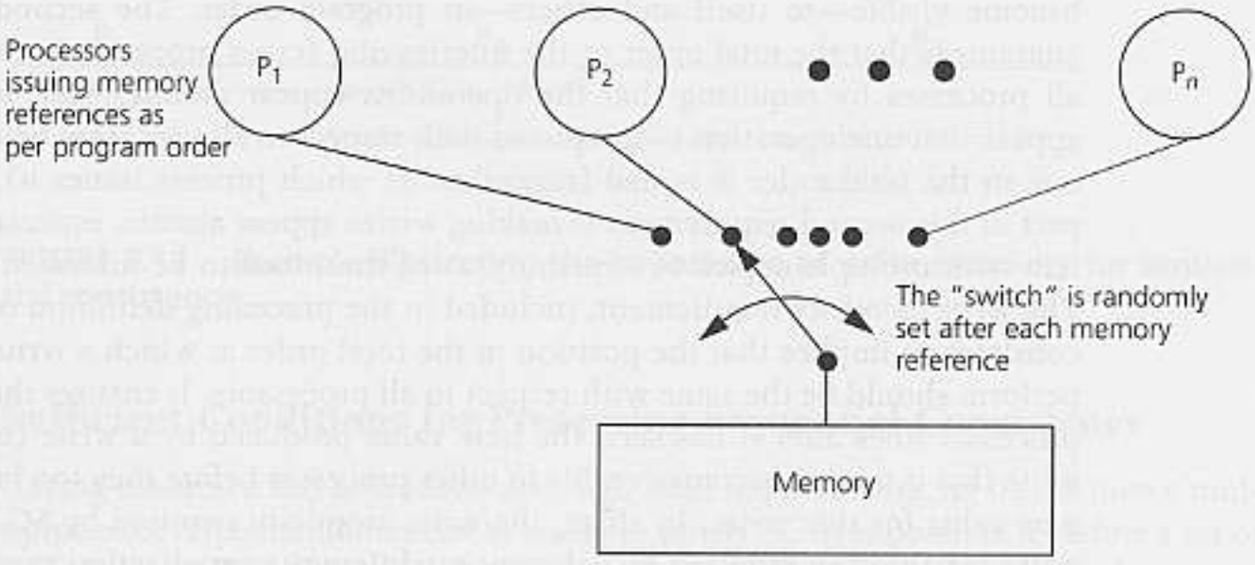

FIGURE 5.10 Programmer's abstraction of the memory subsystem under the sequential consistency model. The model completely hides the underlying concurrency in the memory system hardware (e.g., the possible existence of distributed main memory, the presence of caches and write buffers) from the programmer.

in Exercise 5.6. Note that SC does not obviate the need for synchronization. The reason is that SC allows operations from different processes to be interleaved arbitrarily and does so at the granularity of individual instructions. Synchronization is needed if we want to preserve atomicity (mutual exclusion) across multiple memory operations from a process or if we want to enforce constraints on the interleaving across processes.

The term "program order" also bears some elaboration. Intuitively, *program order* for a process is simply the order in which statements appear according to the source code that the process executes; more specifically, it is the order in which memory operations occur in the assembly code that results from a straightforward translation of source statements one by one to machine instructions. This is not necessarily the order in which an optimizing compiler presents memory operations to the hardware since the compiler may reorder memory operations (within certain constraints, such as preserving dependences to the same location). The programmer has in mind the order of statements in the source program, but the processor sees only the order of the machine instructions. In fact, there is a "program order" at each of the interfaces in the parallel computer architecture—particularly the programming model interface seen by the programmer and the hardware/software interface—and ordering models may be defined at each. Since the programmer reasons with the source program, it makes sense to use this to define program order when discussing memory consistency models; that is, we will be concerned with the consistency model presented by the language and the underlying system to the programmer.

Implementing SC requires that the system (software and hardware) preserve the intuitive constraints defined previously. There are really two constraints. The first is the program order requirement: memory operations of a process must appear to

become visible—to itself and others—in program order. The second constraint guarantees that the total order or the interleaving across processes is consistent for all processes by requiring that the operations appear atomic. That is, it should appear that one operation is completed with respect to all processes before the next one in the total order is issued (regardless of which process issues it). The tricky part of this second requirement is making writes appear atomic, especially in a system with multiple copies of a memory word that need to be informed on a write. The *write atomicity* requirement, included in the preceding definition of sequential consistency, implies that the position in the total order at which a write appears to perform should be the same with respect to all processors. It ensures that nothing a processor does after it has seen the new value produced by a write (e.g., another write that it issues) becomes visible to other processes before they too have seen the new value for that write. In effect, the write atomicity required by SC extends the write serialization required by coherence: while write serialization says that writes to the same location should appear to all processors to have occurred in the same order, write atomicity says that all writes (to any location) should appear to all processors to have occurred in the same order. Example 5.4 shows why write atomicity is important.

EXAMPLE 5.4 Consider the three processes in Figure 5.11. Show how not preserving write atomicity violates sequential consistency.

Answer Since P_2 waits until A becomes 1 and then sets B to 1, and since P_3 waits until B becomes 1 and only then reads the value of A, from transitivity we would infer that P_3 should find the value of A to be 1. If P_2 is allowed to go on past the read of A and write B before it is guaranteed that P_3 has seen the new value of A, then P_3 may read the new value of B but read the old value of A (e.g., from its cache), violating our sequentially consistent intuition. ■

More formally, each process's program order imposes a partial order on the set of all operations; that is, it imposes an ordering on the subset of the operations that are issued by that process. An interleaving of the operations from different processes defines a total order on the set of all operations. Since the exact interleaving is not defined by SC, interleaving the partial (program) orders for different processes may yield a large number of possible total orders. The following definitions therefore apply:

- *Sequentially consistent execution.* An execution of a program is said to be sequentially consistent if the results it produces are the same as those produced by any one of the possible total orders (interleavings) as defined earlier. That is, a total order or interleaving of program orders from processes should exist that yields the same result as the actual execution.
- *Sequentially consistent system.* A system is sequentially consistent if any possible execution on that system is sequentially consistent.

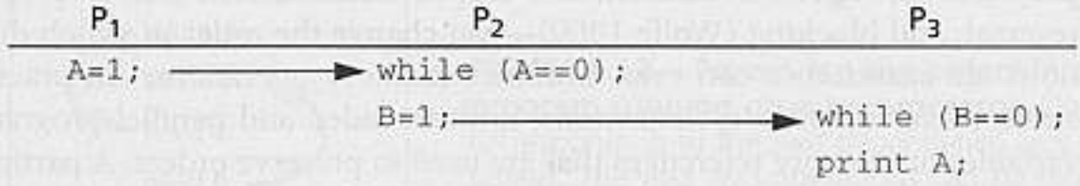

FIGURE 5.11 Example illustrating the importance of write atomicity for sequential consistency

5.2.2 Sufficient Conditions for Preserving Sequential Consistency

Having discussed the definitions and high-level requirements, let us see how a multiprocessor implementation can be made to satisfy SC. It is possible to define a set of sufficient conditions that will guarantee sequential consistency in a multiprocessor—whether bus-based or distributed, cache-coherent or not. The following set, adapted from its original form (Dubois, Scheurich, and Briggs 1986; Scheurich and Dubois 1987), is relatively simple:

1. Every process issues memory operations in program order.
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.
3. After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation. That is, if the write whose value is being returned has performed with respect to this processor (as it must have if its value is being returned), then the processor should wait until the write has performed with respect to all processors.

The third condition is what ensures write atomicity and is quite demanding. It is not a simple local constraint because the read must wait until the logically preceding write has become globally visible. Note that these are sufficient, rather than necessary, conditions. Sequential consistency can be preserved with less serialization in many situations, as we shall see.

With program order defined in terms of the source program, it is important that the compiler should not change the order of memory operations that it presents to the hardware (processor). Otherwise, sequential consistency from the programmer's perspective may be compromised even before the hardware gets involved. Unfortunately, many of the optimizations that are commonly employed in both compilers and processors violate these sufficient conditions. For example, compilers routinely reorder accesses to different locations within a process, so a processor may in fact issue accesses out of the program order seen by the programmer. Explicitly parallel programs use uniprocessor compilers, which are concerned only about preserving dependences to the same location. Advanced compiler optimizations that greatly improve performance—such as common subexpression elimination, constant

propagation, register allocation, and loop transformations like loop splitting, loop reversal, and blocking (Wolfe 1989)—can change the order in which different locations are accessed or can even eliminate memory operations.[2] In practice, to constrain these compiler optimizations, multithreaded and parallel programs annotate variables or memory references that are used to preserve orders. A particularly stringent example is the use of the `volatile` qualifier in a variable declaration, which prevents the variable from being register allocated or any memory operation on the variable from being reordered with respect to operations before or after it in program order. Example 5.5 illustrates these issues.

EXAMPLE 5.5 How would reordering the memory operations in Figure 5.7 affect semantics in a sequential program (only one of the processes running), in a parallel program running on a multiprocessor, and in a threaded program in which the two processes are interleaved on the same processor? How would you solve the problem?

Answer The compiler may reorder the writes to `A` and `flag` with no impact on a sequential program. However, this can violate our intuition for both parallel programs and concurrent (or multithreaded) uniprocessor programs. In the latter case, a context switch can happen between the two reordered writes, so the process switched in may see the update to `flag` without seeing the update to `A`. Similar violations of intuition occur if the compiler reorders the reads of `flag` and `A`. For many compilers, we can avoid these reorderings by declaring the variable `flag` to be of type `volatile integer` instead of just `integer`. Other solutions are also possible and are discussed in Chapter 9. ■

Even if the compiler preserves program order, modern processors use sophisticated mechanisms like write buffers, interleaved memory, pipelining, and out-of-order execution techniques (Hennessy and Patterson 1996). These allow memory operations from a process to issue, execute, and/or complete out of program order. Like compiler optimizations, these architectural optimizations work for sequential programs because the appearance of program order in these programs requires that dependences be preserved only among accesses to the same memory location, as shown in Figure 5.12. The problem in parallel programs is that the out-of-order processing of operations to different shared variables by a process can be detected by other processes.

Preserving the sufficient conditions for SC in multiprocessors is quite a strong requirement since it limits compiler reordering and out-of-order processing techniques. Several weaker consistency models have been proposed and techniques have been developed to satisfy SC while relaxing the sufficient conditions. We will examine these approaches in the context of scalable shared address space machines in Chapter 9. For the purposes of this chapter, we assume the compiler does not reorder memory operations, so the program order that the processor sees is the same as

2. Note that register allocation, performed by modern compilers to eliminate memory operations, can affect coherence itself, not just memory consistency. For the flag synchronization example in Figure 5.7, if the compiler were to register-allocate the `flag` variable for process P_2, the process could end up spinning forever: the cache coherence hardware updates or invalidates only the memory and the caches, not the registers of the machine, so the write propagation property of coherence is violated.

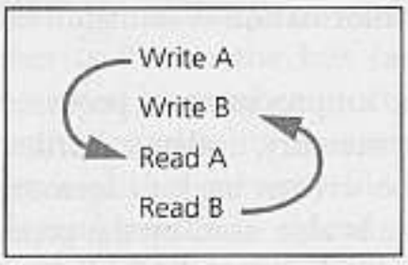

FIGURE 5.12 Preserving the orders in a sequential program running on a uniprocessor. Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

that seen by the programmer. On the hardware side, we assume that the sufficient conditions must be satisfied. To do this, we need mechanisms for a processor to detect completion of its writes so it may proceed past them (completion of reads is easy; a read completes when the data returns to the processor) and mechanisms to satisfy the condition that preserves write atomicity. For all the protocols and systems considered in this chapter, we see how they satisfy coherence (including write serialization), how they can satisfy sequential consistency (in particular, how write completion is detected and write atomicity is guaranteed), and what shortcuts can be taken while still satisfying the sufficient conditions.

For bus-based machines, the serialization imposed by transactions appearing on the shared bus is very useful in ordering memory operations. It is easy to verify that the two-state write-through invalidation protocol discussed previously actually provides sequential consistency—not just coherence—quite easily. The key observation to extend the arguments made for coherence in that system is that writes and read misses to all locations, not just to individual locations, are serialized in bus order. When a read obtains the value of a write, the write is guaranteed to have completed since it caused a previous bus transaction, thus ensuring write atomicity. When a write is performed with respect to any processor, all previous writes in bus order have completed.

5.3 DESIGN SPACE FOR SNOOPING PROTOCOLS

The beauty of snooping-based cache coherence is that the entire machinery for solving a difficult problem boils down to applying a small amount of extra interpretation to events that naturally occur in the system. The processor is completely unchanged. No explicit coherence operations must be inserted in the program. By extending the requirements on the cache controller and exploiting the properties of the bus, the reads and writes that are inherent to the program are used implicitly to keep the caches coherent, and the serialization provided by the bus maintains consistency. Each cache controller observes and interprets the bus transactions generated by others to maintain its internal state. Our initial design point with write-through caches is not very efficient, but we are now ready to study the design space for snooping protocols that make efficient use of the limited bandwidth of the shared bus. All of these use write-back caches, allowing processors to write to different blocks in their local caches concurrently without any bus transactions. Thus,

extra care is required to ensure that enough information is transmitted over the bus to maintain coherence.

Recall that with a write-back cache on a uniprocessor, a processor write miss causes the cache to read the entire block from memory, update a word, and retain the block in *modified* (or *dirty*) state so it may be written back to memory on replacement. In a multiprocessor, this modified state is also used by the protocols to indicate exclusive ownership of the block by a cache. In general, a cache is said to be the *owner* of a block if it must supply the data upon a request for that block (Sweazey and Smith 1986). A cache is said to have an *exclusive* copy of a block if it is the only cache with a valid copy of the block (main memory may or may not have a valid copy). Exclusivity implies that the cache may modify the block without notifying anyone else. If a cache does not have exclusivity, then it cannot write a new value into the block before first putting a transaction on the bus to communicate with others. The writer may have the block in its cache in a valid state, but since a transaction must be generated, it is called a write miss just like a write to a block that is not present or is invalid in the cache. If a cache has the block in modified state, then clearly it is the owner and it has exclusivity. (The need to distinguish ownership from exclusivity will become clear soon.)

On a write miss in an invalidation protocol, a special form of transaction called a *read exclusive* is used to tell other caches about the impending write and to acquire a copy of the block with exclusive ownership. This places the block in the cache in modified state, where it may now be written. Multiple processors cannot write the same block concurrently since this would lead to inconsistent values. The read-exclusive bus transactions generated by their writes will be serialized by the bus, so only one of them can have exclusive ownership of the block at a time. The cache coherence actions are driven by these two types of transactions: read and read exclusive. Eventually, when a modified block is replaced from the cache, the data is written back to memory, but this event is not caused by a memory operation to that block and is almost incidental to the protocol. A block that is not in modified state need not be written back upon replacement and can simply be dropped since memory has the latest copy. Many protocols have been devised for write-back caches, and we examine the basic alternatives.

We also consider update-based protocols. Recall that in update-based protocols, whenever a shared location is written to by a processor, its value is updated in the caches of all other processors holding that memory block.[3] Thus, when these processors subsequently access that block, they can do so from their caches with low latency. The caches of all other processors are updated with a single bus transaction, thus conserving bandwidth when there are multiple sharers. In contrast, with invalidation-based protocols, on a write operation the cache state of that memory block in all other processors' caches is set to invalid, so those processors will have to obtain the block through a miss and hence a bus transaction on their next read.

3. This is a write-broadcast scenario. Read-broadcast designs have also been investigated, in which the cache containing the modified copy flushes it to the bus when it sees a read on the bus, at which point all other copies are updated too.

However, subsequent writes to that block by the same processor do not create further traffic on the bus (as they do with an update protocol) until the block is accessed by another processor. This is attractive when a single processor performs multiple writes to the same memory block before other processors access the contents of that memory block. The detailed trade-offs are more complex, and they depend on the workload offered to the machine; they will be illustrated quantitatively in Section 5.4. In general, invalidation-based strategies have been found to be more robust and are therefore provided as the default protocol by most vendors. Some vendors provide an update protocol as an option to be used for blocks corresponding to selected data structures or pages.

The choices made for the protocol (update versus invalidate) and the caching strategies directly affect the choice of states, the state transition diagram, and the associated actions. Substantial flexibility is available to the computer architect in the design task at this level. Instead of listing all possible choices, let us consider three common coherence protocols that will illustrate the design options.

5.3.1 A Three-State (MSI) Write-Back Invalidation Protocol

The first protocol we consider is a basic invalidation-based protocol for write-back caches. It is very similar to the protocol that was used in the Silicon Graphics 4D series multiprocessor machines (Baskett, Jermoluk, and Solomon 1988). The protocol uses the three states required for any write-back cache in order to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty). Specifically, the states are *modified* (M), *shared* (S), and *invalid* (I). Invalid has the obvious meaning. Shared means the block is present in an unmodified state in this cache, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified, also called dirty, means that only this cache has a valid copy of the block, and the copy in main memory is stale. Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated via a read-exclusive bus transaction. This transaction serves to order the write as well as cause the invalidations and hence ensure that the write becomes visible to others (write propagation).

The processor issues two types of requests: reads (PrRd) and writes (PrWr). The read or write could be to a memory block that exists in the cache or to one that does not. In the latter case, a block currently in the cache will have to be replaced by the newly requested block, and if the existing block is in the modified state, its contents will have to be written back to main memory.

We assume that the bus allows the following transactions:

- *Bus Read* (BusRd): This transaction is generated by a PrRd that misses in the cache, and the processor expects a data response as a result. The cache controller puts the address on the bus and asks for a copy that it does not intend to modify. The memory system (possibly another cache) supplies the data.
- *Bus Read Exclusive* (BusRdX): This transaction is generated by a PrWr to a block that is either not in the cache or is in the cache but not in the modified

state. The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. The memory system (possibly another cache) supplies the data. All other caches are invalidated. Once the cache obtains the exclusive copy, the write can be performed in the cache. The processor may require an acknowledgment as a result of this transaction.

- *Bus Write Back* (BusWB): This transaction is generated by a cache controller on a write back; the processor does not know about it and does not expect a response. The cache controller puts the address and the contents for the memory block on the bus. The main memory is updated with the latest contents.

The bus read exclusive (sometimes called *read-to-own*) is the only new transaction that would not exist except for cache coherence. The new action needed to support write-back protocols is that, in addition to changing the state of cached blocks, a cache controller can intervene in an observed bus transaction and flush the contents of the referenced block from its cache onto the bus rather than allowing the memory to supply the data. Of course, the cache controller can also initiate bus transactions as described above, supply data for write backs, or pick up data supplied by the memory system.

State Transitions

The state transition diagram that governs a block in each cache in this snooping protocol is as shown in Figure 5.13. The states are organized so that the closer the state is to the top, the more tightly the block is bound to that processor. A processor read to a block that is invalid (or not present) causes a BusRd transaction to service the miss. The newly loaded block is *promoted*, moved up in the state diagram, from invalid to the shared state in the requesting cache, whether or not any other cache holds a copy. Any other caches with the block in the shared state observe the BusRd but take no special action, allowing main memory to respond with the data. However, if a cache has the block in the modified state (there can only be one) and it observes a BusRd transaction on the bus, then it must get involved in the transaction since the copy in main memory is stale. This cache flushes the data onto the bus, in lieu of memory, and *demotes* its copy of the block to the shared state (see Figure 5.13). The memory and the requesting cache both pick up the block. This can be accomplished either by a direct cache-to-cache transfer across the bus during this BusRd transaction or by signaling an error on the BusRd transaction and generating a write transaction to update memory. In the latter case, the original cache will eventually retry its request and obtain the block from memory. (It is also possible to have the flushed data picked up only by the requesting cache but not by memory, leaving memory still out-of-date, but this requires more states [Sweazey and Smith 1986].)

Writing into an invalid block is a write miss, which is serviced by first loading the entire block and then modifying the desired bytes within it. The write miss generates a read-exclusive bus transaction, which causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the

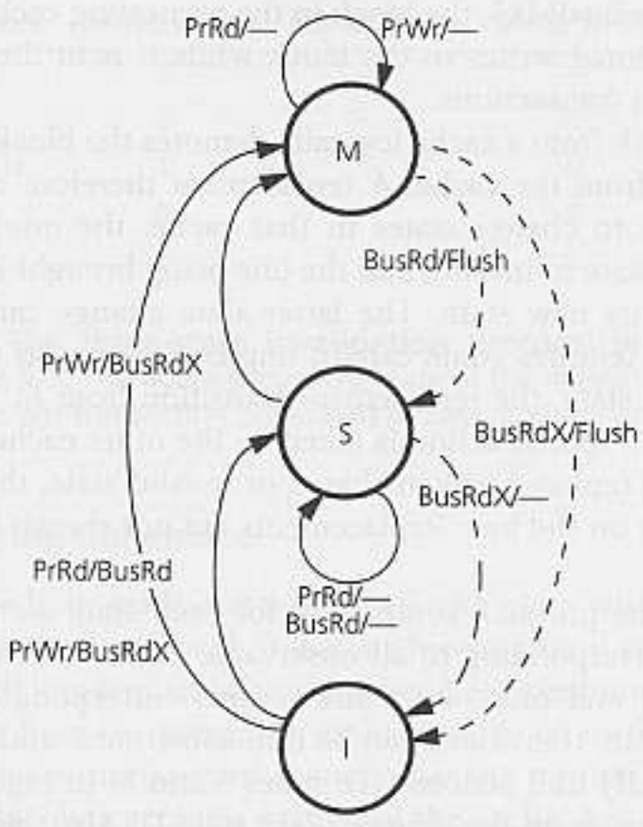

FIGURE 5.13 Basic three-state invalidation protocol. M, S, and I stand for modified, shared, and invalid states, respectively. The notation *A/B* means that if the controller observes the event *A* from the processor side or the bus side, then in addition to the state change, it generates the bus transaction or action *B*. "—" means null action. Transitions due to observed bus transactions are shown in dashed arcs, while those due to local processor actions are shown in bold arcs. If multiple *A/B* pairs are associated with an arc, it simply means that multiple inputs can cause the same state transition. For completeness, we should specify actions from each state corresponding to each observable event. If such transitions are not shown, it means that they are uninteresting and no action needs to be taken. Replacements and the write backs they may cause are not shown in the diagram for simplicity.

block. The block of data returned by the read exclusive is promoted to the modified state, and the desired bytes are then written into it. If another cache later requests exclusive access, then in response to its BusRdX transaction this block will be invalidated (demoted to the invalid state) after flushing the exclusive copy to the bus.

The most interesting transition occurs when writing into a shared block. As discussed earlier, this is treated essentially like a write miss, using a read-exclusive bus transaction to acquire exclusive ownership; we refer to it as a write miss throughout the book. The data that comes back in the read exclusive can be ignored in this case, unlike when writing to an invalid or not present block, since it is already in the cache. In fact, a common optimization to reduce data traffic in bus protocols is to introduce a new transaction, called a *bus upgrade* or BusUpgr, for this situation. A BusUpgr obtains exclusive ownership just like a BusRdX, by causing other copies to be invalidated, but it does not cause main memory or any other device to respond with the data for the block. Regardless of whether a BusUpgr or a BusRdX is used

(let us continue to assume BusRdX), the block in the requesting cache transitions to the modified state. Additional writes to the block while it is in the modified state generate no additional bus transactions.

A replacement of a block from a cache logically demotes the block to invalid (not present) by removing it from the cache. A replacement therefore causes the state machines for two blocks to change states in that cache: the one being replaced changes from its current state to invalid, and the one being brought in changes from invalid (not present) to its new state. The latter state change cannot take place before the former, which requires some care in implementation. If the block being replaced was in modified state, the replacement transition from M to I generates a write-back transaction. No special action is taken by the other caches on this transaction. If the block being replaced was in shared or invalid state, then it itself does not cause any transaction on the bus. Replacements are not shown in the state diagram for simplicity.

Note that to specify the protocol completely, for each state we must have outgoing arcs with labels corresponding to all observable events (the inputs from the processor and bus sides) and must show the actions corresponding to them. Of course, the actions and state transitions can be null sometimes, and in that case we may either explicitly specify null actions (see states S and M in Figure 5.13), or we may simply omit those arcs from the diagram (see state I). Also, since we treat the not-present state as invalid, when a new block is brought into the cache on a miss, the state transitions are performed as if the previous state of the block was invalid. Example 5.6 illustrates how the state transition diagram is interpreted.

EXAMPLE 5.6 Using the MSI protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5.3.

Answer The results are shown in Figure 5.14. ■

With write-back protocols, a block can be written many times before the memory is actually updated. A read may obtain data not from memory but rather from a writer's cache, and in fact it may be this read rather than a replacement that causes memory to be updated. In addition, write hits do not appear on the bus, so the concept of a write being performed with respect to other processors is a little different. In fact, to say that a write is being performed means that the write is being "made visible." A write to a shared or invalid block is made visible by the bus read-exclusive transaction it triggers. The writer will "observe" the data in its cache after this transaction. The write will be made visible to other processors by the invalidations that the read exclusive generates, and those processors will experience a cache miss before actually observing the value written. Write hits to a modified block are visible to other processors but again are observed by them only after a miss through a bus transaction. Thus, in the MSI protocol, the write to a nonmodified block is performed or made visible when the BusRdX transaction occurs, and the write to a modified block is made visible when the block is updated in the writer's cache.

Processor Action	State in P_1	State in P_2	State in P_3	Bus Action	Data Supplied By
1. P_1 reads u	S	–	–	BusRd	Memory
2. P_3 reads u	S	–	S	BusRd	Memory
3. P_3 writes u	I	–	M	BusRdX	Memory
4. P_1 reads u	S	–	S	BusRd	P_3 cache
5. P_2 reads u	S	S	S	BusRd	Memory

FIGURE 5.14 The three-state invalidation protocol in action for processor transactions shown in Figure 5.3. The figure shows the state of the relevant memory block at the end of each processor action, the bus transaction generated (if any), and the entity supplying the data.

Satisfying Coherence

Since both reads and writes can take place without generating bus transactions in a write-back protocol, it is not obvious that it satisfies the conditions for coherence, much less sequential consistency. Let's examine coherence first. Write propagation is clear from the preceding discussion, so let us focus on write serialization. The read-exclusive transaction ensures that the writing cache has the only valid copy when the block is actually written in the cache, just like a write transaction in the write-through protocol. It is followed immediately by the corresponding write being performed in the cache before any other bus transactions are handled by that cache controller, so it is ordered in the same way for all processors (including the writer) with respect to other bus transactions. The only difference from a write-through protocol, with regard to ordering operations to a location, is that not all writes generate bus transactions. However, the key here is that between two transactions for that block that do appear on the bus, only one processor can perform such write hits; this is the processor (say, P) that performed the most recent read-exclusive bus transaction w for the block. In the serialization, this sequence of write hits therefore appears (in program order) between w and the next bus transaction for that block. Reads by processor P will clearly see them in this order with respect to other writes. For a read by another processor, there is at least one bus transaction for that block that separates the completion of that read from the completion of these write hits. That bus transaction ensures that that read also sees the writes in the consistent serial order. Thus, reads by all processors see all writes in the same order.

Satisfying Sequential Consistency

To see how SC is satisfied, let us first appeal to the definition itself and see how a consistent global interleaving of all memory operations may be constructed. As with write-through caches, the serial arbitration for the bus in fact defines a total order on bus transactions for all blocks, not just those for a single block. All cache controllers observe read and read-exclusive bus transactions in the same order and perform invalidations in this order. Between consecutive bus transactions, each processor

performs a sequence of memory operations (read and write hits) in program order. Thus, any execution of a program defines a natural partial order:

> A memory operation M_j is subsequent to operation M_i if (1) the operations are issued by the same processor and M_j follows M_i in program order, or (2) M_j generates a bus transaction that follows the memory operation for M_i.

This partial order looks graphically like that of Figure 5.6, except the local sequence within a segment has writes as well as reads and both read-exclusive and read bus transactions play important roles in establishing the orders. Between bus transactions, any interleaving of the sequences of local operations (hits) from different processors leads to a consistent total order. For writes that occur in the same segment between bus transactions, a processor will observe the writes by other processors ordered by bus transactions that it generates, and its own writes ordered by program order.

We can also see how SC is satisfied in terms of the sufficient conditions. Write completion is detected when the read-exclusive bus transaction occurs on the bus and the write is performed in the cache. The read completion condition, which provides write atomicity, is met because a read either (1) causes a bus transaction that follows that of the write whose value is being returned, in which case the write must have completed globally before the read; (2) follows such a read by the same processor in program order; or (3) follows in program order on the same processor that performed the write, in which case the processor has already waited for the write to complete (become visible) globally. Thus, all the sufficient conditions are easily guaranteed. We return to this topic when we discuss implementing protocols in Chapter 6.

Lower-Level Design Choices

To illustrate some of the implicit design choices that have been made in the protocol, let us examine more closely the transition from the M state when a BusRd for that block is observed. In Figure 5.13, we transition to state S and flush the contents of the memory block to the bus. Although it is imperative that the contents are placed on the bus, we could instead have transitioned to state I, thus giving up the block entirely. The choice of going to S versus I reflects the designer's assertion that the original processor is more likely to continue reading the block than the new processor is to write to the memory block. Intuitively, this assertion holds for mostly read data, which is common in many programs. However, a common case where it does not hold is for a flag or buffer that is used to transfer information back and forth between processes: one processor writes it, the other reads it and modifies it, then the first reads it and modifies it, and so on. Accumulations into a shared counter exhibit similar *migratory* behavior across multiple processors. The problem with betting on read sharing in these cases is that every write has to first generate an invalidation, thereby increasing its latency. Indeed, the coherence protocol used in the early Synapse multiprocessor made the alternate choice of going directly from M to I state on a BusRd, thus betting the migratory pattern would be more frequent.

Some machines (Sequent Symmetry model B and the MIT Alewife) attempt to adapt the protocol when such a migratory access pattern is observed (Cox and Fowler 1993; Dahlgren, Dubois, and Stenstrom 1994). These choices can affect the performance of the memory system, as we see later in the chapter.

5.3.2 A Four-State (MESI) Write-Back Invalidation Protocol

A concern arises with our MSI protocol if we consider a sequential application running on a multiprocessor. Such multiprogrammed use in fact constitutes the most common workload on small-scale multiprocessors. When the process reads in and modifies a data item, in the MSI protocol two bus transactions are generated even though there are never any sharers. The first is a BusRd that gets the memory block in S state, and the second is a BusRdX (or BusUpgr) that converts the block from S to M state. By adding a state that indicates that the block is the only (exclusive) copy but is not modified and by loading the block in this state, we can save the latter transaction since the state indicates that no other processor is caching the block. This new state, called *exclusive-clean* or *exclusive-unowned* (or even simply "exclusive"), indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state, the cache can perform a write and move to the modified state without further bus transactions; but it does not imply ownership (memory has a valid copy), so unlike the modified state, the cache need not reply upon observing a request for the block. Variants of this MESI protocol are used in many modern microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors. It was first published by researchers at the University of Illinois at Urbana-Champaign (Papamarcos and Patel 1984) and is often referred to as the Illinois protocol (Archibald and Baer 1986).

The MESI protocol thus consists of four states: modified (M) or dirty, exclusive-clean (E), shared (S), and invalid (I). M and I have the same semantics as before. E, the exclusive-clean or exclusive state, means that only one cache (this cache) has a copy of the block and it has not been modified (i.e., the main memory is up-to-date). S means that potentially two or more processors have this block in their cache in an unmodified state. The bus transactions and actions needed are very similar to those for the MSI protocol.

State Transitions

When the block is first read by a processor, if a valid copy exists in another cache, then it enters the processor's cache in the S state, as usual. However, if no other cache has a copy at the time (for example, in a sequential application), it enters the cache in the E state. When that block is written by the same processor, it can directly transition from E to M state without generating another bus transaction since no other cache has a copy. If another cache had obtained a copy in the meantime, the state of the block would have been demoted from E to S by the snooping protocol.

This protocol places a new requirement on the physical interconnect of the bus. An additional signal, called the shared signal (S), must be available to the controllers in order to determine on a BusRd if any other cache currently holds the data. During the address phase of the bus transaction, all caches determine if they contain the requested block and, if so, assert the shared signal. This signal is a wired-OR line, so the controller making the request can observe whether any other processors are caching the referenced memory block and can thereby decide whether to load a requested block in the E state or the S state.

Figure 5.15 shows a state transition diagram for a MESI protocol, still assuming that the BusUpgr transaction is not used. The notation BusRd(S) means that the bus read transaction caused the shared signal S to be asserted; BusRd($\bar{S}$) means S was unasserted. A plain BusRd means that we don't care about the value of S for that transition. A write to a block in any state will promote the block to the M state, but if it was in the E state, then no bus transaction is required. Observing a BusRd will demote a block from E to S since now another cached copy exists. As usual, observing a BusRd will demote a block from M to S state and will also cause the block to be flushed onto the bus; here too, the block may be picked up only by the requesting cache and not by main memory, but this may require additional states beyond MESI. (A fifth, *owned* state may be added, which indicates that even though other shared copies of the block may exist, this cache [instead of main memory] is responsible for supplying the data when it observes a relevant bus transaction. This leads to a five-state MOESI protocol [Sweazey and Smith 1986].) Notice that it is possible for a block to be in the S state even if no other copies exist since copies may be replaced ($S \rightarrow I$) without notifying other caches. The arguments for satisfying coherence and sequential consistency are the same as in the MSI protocol.

Lower-Level Design Choices

An interesting question for bus-based protocols is who should supply the block for a BusRd transaction when both the memory and another cache have a copy of it. In the original (Illinois) version of the MESI protocol, the cache rather than main memory supplied the data—a technique called *cache-to-cache sharing*. The argument for this approach was that caches, being constructed out of SRAM rather than DRAM, could supply the data more quickly. However, this advantage is not necessarily present in modern bus-based machines, in which intervening in another processor's cache to obtain data may be more expensive than obtaining the data from main memory. Cache-to-cache sharing also adds complexity to a bus-based protocol: main memory must wait until it is certain that no cache will supply the data before driving the bus, and if the data resides in multiple caches, then a selection algorithm is needed to determine which one will provide the data. On the other hand, this technique is useful for multiprocessors with physically distributed memory (as we see in Chapter 8) because the latency to obtain the data from a nearby cache may be much smaller than that for a faraway memory unit. This effect can be especially important for machines constructed as a network of SMP nodes because caches

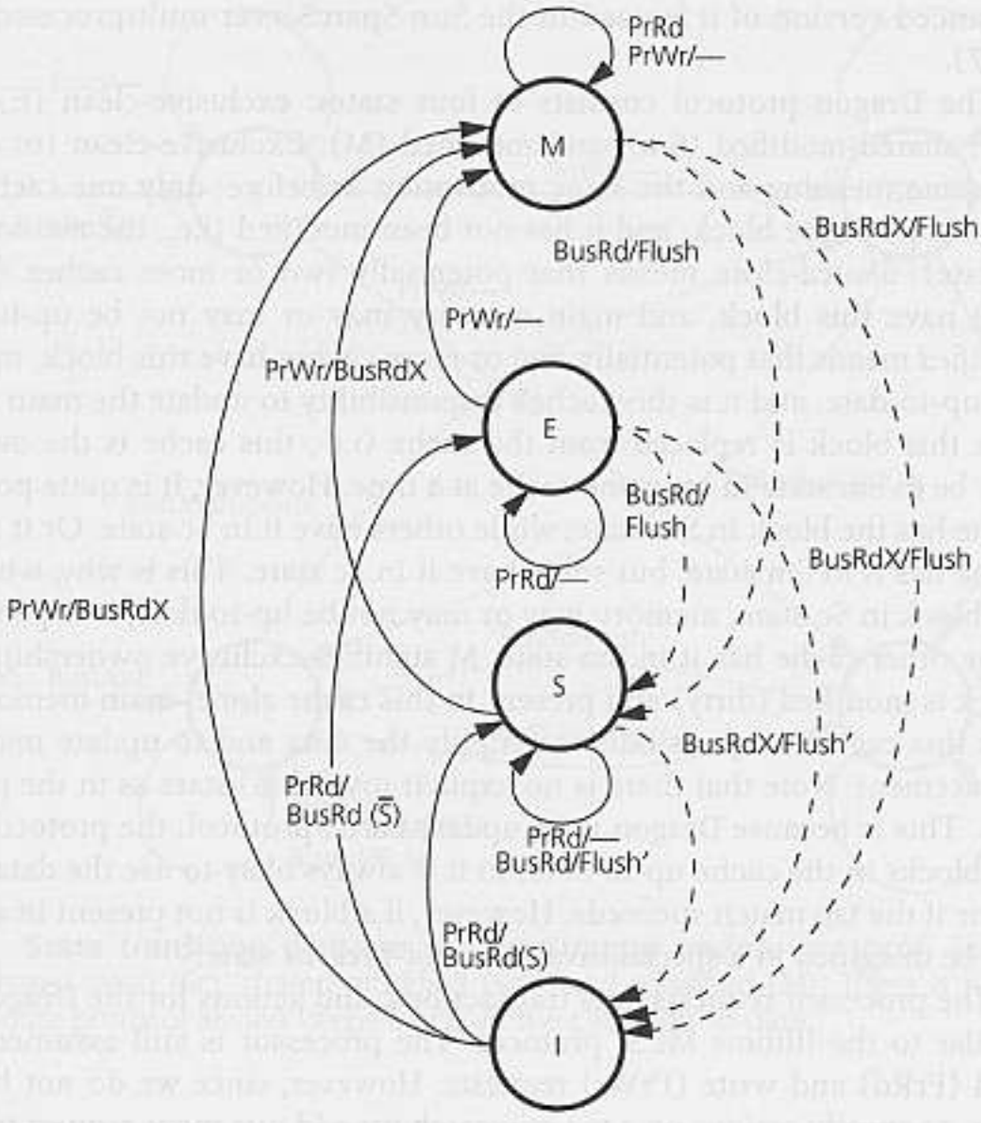

FIGURE 5.15 State transition diagram for the Illinois MESI protocol. MESI stands for the modified (dirty), exclusive, shared, and invalid states, respectively. The notation is the same as that in Figure 5.13. The E state helps reduce bus traffic for sequential programs where data is not shared. Whenever feasible, the Illinois version of the MESI protocol makes caches, rather than main memory, supply data for BusRd and BusRdX transactions. Since multiple processors may have a copy of the memory block in their cache, we need to select only one to supply the data on the bus. Flush' is true only for that processor; the remaining processors take their usual action (invalidation or no action). In general, Flush' in a state diagram indicates that the block is flushed only if cache-to-cache sharing is in use and then only by the cache that is responsible for supplying the data.

within the requestor's SMP node may supply the data. The Stanford DASH multiprocessor (Lenoski et al. 1993) used such cache-to-cache transfers for this reason.

5.3.3 A Four-State (Dragon) Write-Back Update Protocol

Let us now examine a basic update-based protocol for write-back caches. This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessor system (McCreight 1984; Thacker, Stewart, and Satterthwaite 1988), and an

enhanced version of it is used in the Sun SparcServer multiprocessors (Catanzaro 1997).

The Dragon protocol consists of four states: exclusive-clean (E), shared-clean (Sc), shared-modified (Sm), and modified (M). Exclusive-clean (or exclusive) has the same meaning and the same motivation as before: only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). *Shared-clean* means that potentially two or more caches (including this one) have this block, and main memory may or may not be up-to-date. *Shared-modified* means that potentially two or more caches have this block, main memory is not up-to-date, and it is this cache's responsibility to update the main memory at the time this block is replaced from the cache (i.e., this cache is the owner). A block may be in Sm state in only one cache at a time. However, it is quite possible that one cache has the block in Sm state, while others have it in Sc state. Or it may be that no cache has it in Sm state, but some have it in Sc state. This is why, when a cache has the block in Sc state, memory may or may not be up-to-date; it depends on whether some other cache has it in Sm state. M signifies exclusive ownership as before: the block is modified (dirty) and present in this cache alone, main memory is stale, and it is this cache's responsibility to supply the data and to update main memory on replacement. Note that there is no explicit invalid (I) state as in the previous protocols. This is because Dragon is an update-based protocol; the protocol always keeps the blocks in the cache up-to-date, so it is always okay to use the data present in the cache if the tag match succeeds. However, if a block is not present in a cache at all, it can be imagined in a special invalid or not-present state.[4]

The processor requests, bus transactions, and actions for the Dragon protocol are similar to the Illinois MESI protocol. The processor is still assumed to issue only read (PrRd) and write (PrWr) requests. However, since we do not have an invalid state, to specify actions on a tag mismatch we add two more request types: processor read miss (PrRdMiss) and write miss (PrWrMiss). As for bus transactions, we have bus read (BusRd), bus write back (BusWB), and a new transaction called bus update (BusUpd). The BusRd and BusWB transactions have the usual semantics. The BusUpd transaction takes the specific word (or bytes) written by the processor and broadcasts it on the bus so that all other processors' caches can update themselves. By broadcasting only the contents of the specific modified word rather than the whole cache block, it is hoped that the bus bandwidth is more efficiently utilized. (See Exercise 5.4 for reasons why this may not always be the case.) As in the MESI protocol, to support the E state, a shared signal (S) is available to the cache controller. Finally, the only new capability needed is for the cache controller to update a locally cached memory block (labeled an Update action) with the contents that are being broadcast on the bus by a relevant BusUpd transaction.

4. Logically, there is another state as well, but it is rather crude and is used to bootstrap the protocol. A "miss mode" bit is provided with each cache line to force a miss when that block is accessed. Initialization software reads data into every line in the cache with the miss mode bit turned on to ensure that the processor will miss the first time it references a block that maps to that line. After this first miss, the miss mode bit is turned off and the cache operates normally.

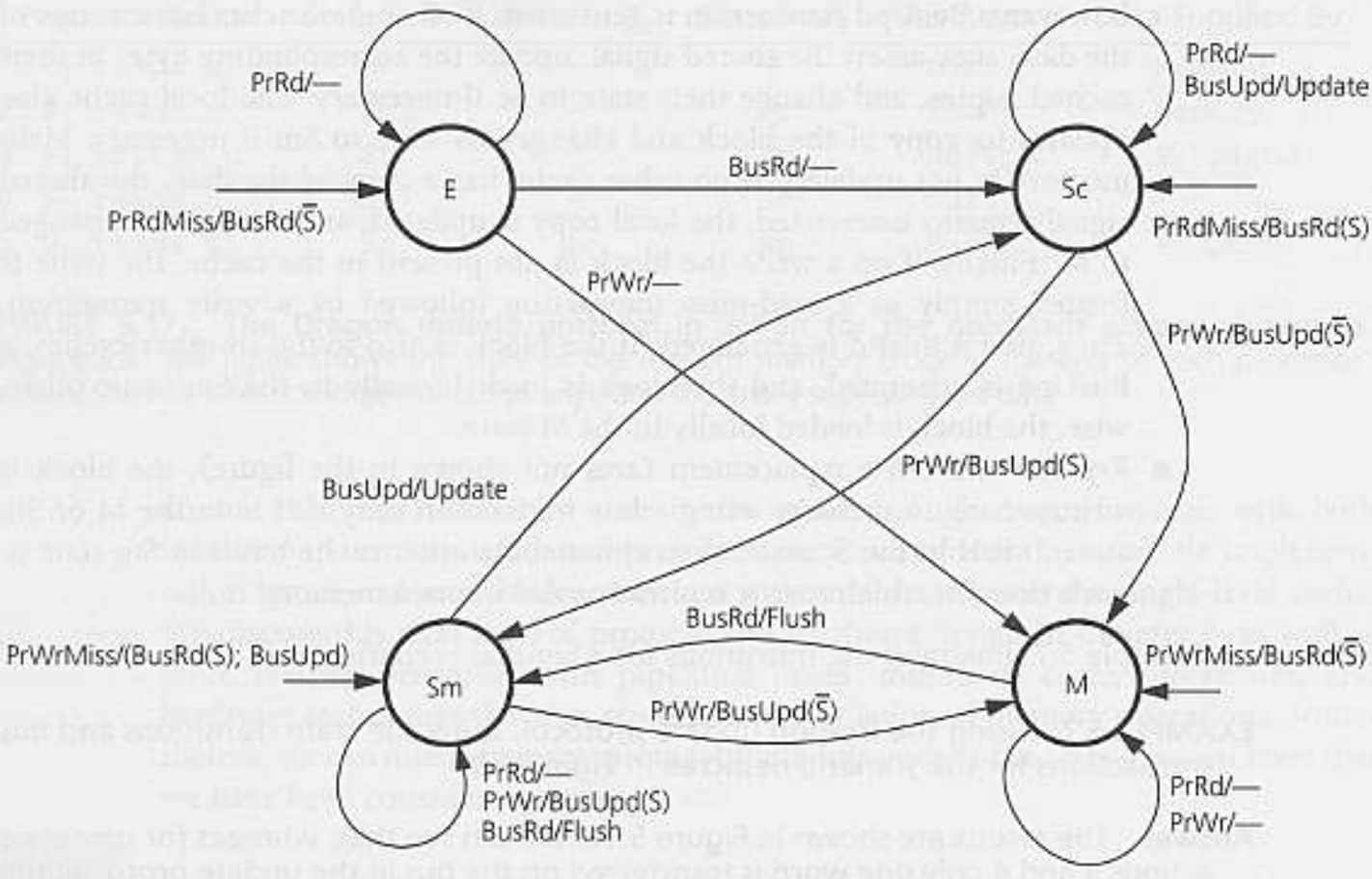

FIGURE 5.16 State transition diagram for the Dragon update protocol. The four states are exclusive (E), shared-clean (Sc), shared-modified (Sm), and modified (M). There is no invalid (I) state because the update protocol always keeps blocks in the cache up-to-date.

State Transitions

Figure 5.16 shows the state transition diagram for the Dragon update protocol. To take a processor-centric view, we can explain the diagram in terms of actions taken when a cache incurs a read miss, a write (hit or miss), or a replacement (no action is ever taken on a read hit).

- *Read miss:* A BusRd transaction is generated. Depending on the status of the shared signal (S), the block is loaded in the E or Sc state in the local cache. If the block is in M or Sm states in one of the other caches, that cache asserts the shared signal and supplies the latest data for that block on the bus, and the block is loaded in the local cache in Sc state. If the other cache had it in state M, it changes its state to Sm. If the block is in Sc state in other caches, memory supplies the data, and it is loaded in Sc state. If no other cache has a copy, then the shared line remains unasserted, the data is supplied by the main memory, and the block is loaded in the local cache in E state.
- *Write:* If the block is in the M state in the local cache, then no action needs to be taken. If the block is in the E state in the local cache, then it changes to M state and again no further action is needed. If the block is in Sc or Sm state,

however, a BusUpd transaction is generated. If any other caches have a copy of the data, they assert the shared signal, update the corresponding bytes in their cached copies, and change their state to Sc if necessary. The local cache also updates its copy of the block and changes its state to Sm if necessary. Main memory is not updated. If no other cache has a copy of the data, the shared signal remains unasserted, the local copy is updated, and the state is changed to M. Finally, if on a write the block is not present in the cache, the write is treated simply as a read-miss transaction followed by a write transaction. Thus, first a BusRd is generated. If the block is also found in other caches, a BusUpd is generated, and the block is loaded locally in the Sm state; otherwise, the block is loaded locally in the M state.

- *Replacement:* On a replacement (arcs not shown in the figure), the block is written back to memory using a bus transaction only if it is in the M or Sm state. If it is in the Sc state, then either some other cache has it in Sm state or none does, in which case it is already valid in main memory.

Example 5.7 illustrates the transitions for a familiar scenario.

EXAMPLE 5.7 Using the Dragon update protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5.3.

Answer The results are shown in Figure 5.17. We can see that, whereas for processor actions 3 and 4 only one word is transferred on the bus in the update protocol, the whole memory block is transferred twice in the invalidation-based protocol. Of course, it is easy to construct scenarios in which the invalidation protocol does much better than the update protocol, and we discuss the detailed trade-offs in Section 5.4. ■

Lower-Level Design Choices

Again, many implicit design choices have been made in this protocol. For example, it is feasible to eliminate the shared-modified state. In fact, the update protocol used in the DEC Firefly multiprocessor does exactly that. The rationale is that every time the BusUpd transaction occurs, main memory can also update its contents along with the other caches holding that block; therefore, shared clean suffices, and a shared-modified state is not needed. The Dragon protocol is instead based on the assumption that the SRAM caches are much quicker to update than the DRAM main memory, so it is inappropriate to wait for main memory to be updated on all BusUpd transactions. Another subtle choice relates to the action taken on cache replacements. When a shared-clean block is replaced, should other caches be informed of that replacement via a bus transaction so that if only one cache remains with a copy of the memory block, it can change its state to exclusive or modified? The advantage of doing this would be that the bus transaction upon the replacement might not be in the critical path of a memory operation, whereas the later bus transaction that it saves might be.

Since all writes appear on the bus in an update protocol, write serialization, write completion detection, and write atomicity are all quite straightforward with a simple

Processor Action	State in P_1	State in P_2	State in P_3	Bus Action	Data Supplied By
1. P_1 reads u	E	–	–	BusRd	Memory
2. P_3 reads u	Sc	–	Sc	BusRd	Memory
3. P_3 writes u	Sc	–	Sm	BusUpd	P_3 cache
4. P_1 reads u	Sc	–	Sm	null	–
5. P_2 reads u	Sc	Sc	Sm	BusRd	P_3 cache

FIGURE 5.17 The Dragon update protocol in action for the processor actions shown in Figure 5.3. The figure shows the state of the relevant memory block at the end of each processor action, the bus transaction generated (if any), and the entity supplying the data.

atomic bus, a lot like they were in the write-through case. However, with both invalidation- and update-based protocols, we must address many subtle implementation issues and race conditions, even with an atomic bus and a single-level cache. We discuss this next level of protocol and hardware design in Chapter 6, as well as more realistic scenarios with pipelined buses, multilevel cache hierarchies, and hardware techniques that can reorder the completion of memory operations. Nonetheless, we can quantify many protocol trade-offs even at the state diagram level that we have been considering so far.

5.4 ASSESSING PROTOCOL DESIGN TRADE-OFFS

Like any other complex system, the design of a multiprocessor requires many interrelated decisions to be made. Even when a processor has been picked, we must decide on the maximum number of processors to be supported by the system, various parameters of the cache hierarchy (e.g., number of levels in the hierarchy, and for each level the cache size, associativity, block size, and whether the cache is write through or write back), the design of the bus (e.g., width of the data and address buses, the bus protocol), the design of the memory system (e.g., interleaved memory banks or not, width of memory banks, size of internal buffers), and the design of the I/O subsystem. Many of the issues are similar to those in uniprocessors (Smith 1982) but accentuated. For example, a write-through cache standing before the bus may be a poor choice for multiprocessors because the bus bandwidth is shared by many processors, and memory may need to be more greatly interleaved because it services cache misses from multiple processors. Greater cache associativity may also be useful in reducing conflict misses that generate bus traffic.

The cache coherence protocol is a crucial new design issue for a multiprocessor. It includes protocol class (invalidation or update), protocol states and actions, and lower-level implementation trade-offs. Protocol decisions interact with all the other design issues. On the one hand, the protocol influences the extent to which the latency and bandwidth characteristics of system components are stressed; on the other, the performance characteristics as well as the organization of the memory and communication architecture influence the choice of protocols. As discussed in

Chapter 4, these design decisions need to be evaluated relative to the behavior of real programs. Such evaluation was very common in the late 1980s, albeit using an immature set of parallel programs as workloads (Archibald and Baer 1986; Agarwal and Gupta 1988; Eggers and Katz 1988, 1989a, 1989b).

Making design decisions in real systems is part art and part science. The art draws on the past experience, intuition, and aesthetics of the designers, and the science is based in workload-driven evaluation. The goals are usually to meet a cost-performance target and to have a balanced system, so that no individual resource is a performance bottleneck yet each resource has only minimal excess capacity. This section illustrates some key protocol trade-offs by putting the workload-driven evaluation methodology from Chapter 4 into action.

5.4.1 Methodology

The basic strategy is as follows. The workload is executed on a simulator of a multiprocessor architecture, as described in Chapter 4. By observing the state transitions encountered in the simulator, we can determine the frequency of various events such as cache misses and bus transactions. We can then evaluate the effect of protocol choices in terms of other design parameters such as latency and bandwidth requirements.

Choosing parameters according to the methodology of Chapter 4, this section first establishes the basic state transition characteristics generated by the set of applications for the four-state Illinois MESI protocol. It then illustrates how to use these frequency measurements to obtain a preliminary quantitative analysis of the design trade-offs raised by the example protocols above, such as the use of the exclusive state in the MESI protocol and the use of BusUpgr rather than BusRdX transactions for the S → M transition. This section also illustrates more traditional design issues, such as how the cache block size—the granularity of both coherence and communication—impacts the latency and bandwidth needs of the applications. To understand this effect, we classify cache misses into categories such as cold, capacity, and sharing misses, examine the effect of block size on each category, and explain the results in light of application characteristics. Finally, this understanding of the applications is used to illustrate the trade-offs between invalidation-based and update-based protocols, again in light of latency and bandwidth implications.

The analysis in this section is based on the frequency of various important events, not on the absolute times taken or, therefore, the performance. This approach is common in studies of cache architecture because the results transcend particular system implementations and technology assumptions. However, it should be viewed as only a preliminary analysis since many detailed factors that might affect the performance trade-offs in real systems are abstracted away. For example, measuring state transitions provides a means of calculating miss rates and bus traffic, but realistic values for latency, overhead, and occupancy are needed to translate the rates into the actual bandwidth requirements imposed on the system. To obtain an estimate of bandwidth requirements, we may artificially assume that every reference takes a fixed number of cycles to complete. However, the bandwidth requirements them-

selves do not translate into performance directly but only indirectly by increasing the cost of misses due to contention. Contention is very difficult to estimate because it depends on the timing parameters used and on the burstiness of the traffic, which is not captured by the frequency measurements. Contention, timing, and hence performance are also affected by lower-level interactions with hardware structures (like queues and buffers) and policies.

The simulations used in this section do not model contention. Instead, they use a simple PRAM cost model: all memory operations are assumed to complete in the same amount of time (here a single cycle) regardless of whether they hit or miss in the cache. There are three main reasons for this. First, the focus is on understanding inherent protocol behavior and trade-offs in terms of event frequencies, not so much on performance. Second, since we are experimenting with different cache block sizes and organizations, we would like the interleaving of references from application processes on the simulator to be the same regardless of these choices; that is, all protocols and block sizes should see the same trace of references. With the execution-driven rather than trace-driven simulation we use, this is only possible if we make the cost of every memory operation the same in the simulations. Otherwise, if a reference misses with a small cache block but hits with a larger one, for example, then it will be delayed by different amounts in the interleaving in the two cases. It would therefore be difficult to determine which effects are inherently due to the protocol and which are due to the particular parameter values chosen. Third, realistic simulations that model contention take much more time. The disadvantage of using this simple model even to measure frequencies is that the timing model may affect some of the frequencies we observe; however, this effect is small for the applications we study.

The illustrative workloads we use are the six parallel programs (from the SPLASH-2 suite) and one multiprogrammed workload described in Chapters 3 and 4. The parallel programs run in batch mode with exclusive access to the machine and do not include operating system activity in the simulations, whereas the multiprogrammed workload includes operating system activity. The number of applications used is relatively small, but the applications are primarily for illustration as discussed in Chapter 4; the emphasis here is on choosing programs that represent important classes of computation and with widely varying characteristics. The frequencies of basic operations for the applications appear in Table 4.1. We now study them in more detail to assess design trade-offs in cache coherency protocols.

5.4.2 Bandwidth Requirement under the MESI Protocol

We begin by using the default 1-MB, single-level caches per processor, as discussed in Chapter 4. These are large enough to hold the important working sets for the default problem sizes, which is a realistic scenario for all applications. We use four-way set associativity (with LRU replacement) to reduce conflict misses and a 64-byte cache block size for realism. Driving the workloads through a cache simulator that models the Illinois MESI protocol generates the state transition frequencies shown in Table 5.1. The data is presented as the number of state transitions of a particular type per 1,000 references issued by the processors. Note in the table that a new state,

Table 5.1 State Transitions per 1,000 Data Memory References Issued by the Applications

Application			To				
			NP	I	E	S	M
Barnes-Hut	From	NP	0	0	0.0011	0.0362	0.0035
		I	0.0201	0	0.0001	0.1856	0.0010
		E	0.0000	0.0000	0.0153	0.0002	0.0010
		S	0.0029	0.2130	0	97.1712	0.1253
		M	0.0013	0.0010	0	0.1277	902.782
LU	From	NP	0	0	0.0000	0.6593	0.0011
		I	0.0000	0	0	0.0002	0.0003
		E	0.0000	0	0.4454	0.0004	0.2164
		S	0.0339	0.0001	0	302.702	0.0000
		M	0.0001	0.0007	0	0.2164	697.129
Ocean	From	NP	0	0	1.2484	0.9565	1.6787
		I	0.6362	0	0	1.8676	0.0015
		E	0.2040	0	14.0040	0.0240	0.9955
		S	0.4175	2.4994	0	134.716	2.2392
		M	2.6259	0.0015	0	2.2996	843.565
Radiosity	From	NP	0	0	0.0068	0.2581	0.0354
		I	0.0262	0	0	0.5766	0.0324
		E	0	0.0003	0.0241	0.0001	0.0060
		S	0.0092	0.7264	0	162.569	0.2768
		M	0.0219	0.0305	0	0.3125	839.507
Radix	From	NP	0	0	0.004746	3.524705	11.41111
		I	0.130988	0	0	1.108079	4.57868
		E	0.000759	0.002848	0.080301	0	0.00019
		S	0.029804	1.120988	0	178.1932	0.817818
		M	0.044232	11.53127	0	4.03157	802.282

continued

Table 5.1 State Transitions per 1,000 Data Memory References Issued by the Applications

Application			To				
			NP	I	E	S	M
Raytrace	From	NP	0	0	1.3358	1.5486	0.0026
		I	0.0242	0	0.0000	0.3403	0.0000
		E	0.8663	0	29.0187	0.3639	0.0175
		S	1.1181	0.3740	0	310.949	0.2898
		M	0.0559	0.0001	0	0.2970	661.011
Multiprog User Data References	From	NP	0	0	0.1675	0.5253	0.1843
		I	0.2619	0	0.0007	0.0072	0.0013
		E	0.0729	0.0008	11.6629	0.0221	0.0680
		S	0.3062	0.2787	0	214.6523	0.2570
		M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	From	NP	0	0	3.2709	15.7722	0
		I	0	0	0	0	0
		E	1.3029	0	46.7898	1.8961	0
		S	16.9032	0	0	981.2618	0
		M	0	0	0	0	0
Multiprog Kernel Data References	From	NP	0	0	1.0241	1.7209	4.0793
		I	1.3950	0	0.0079	1.1495	0.1153
		E	0.5511	0.0063	55.7680	0.0999	0.3352
		S	1.2740	2.0514	0	393.5066	1.7800
		M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	From	NP	0	0	2.1799	26.5124	0
		I	0	0	0	0	0
		E	0.8829	0	5.2156	1.2223	0
		S	24.6963	0	0	1,075.2158	0
		M	0	0	0	0	0

The data assumes 16 processors (except for Multiprog, which is for 8 processors), 1-MB four-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

NP (not present), is introduced. This addition helps clarify transitions where, on a cache miss, one block is replaced (creating a transition from one of I, E, S, or M to NP) and a new block is brought in (creating a transition from NP to one of I, E, S, or M). The sum of state transitions can be greater than 1,000 even though we are presenting averages per 1,000 references because some references cause multiple state transitions. For example, a write miss can cause two transitions in the local processor's cache (e.g., S → NP for the old block and NP → M for the incoming block), in addition to transitions in other caches due to invalidations (I/E/S/M → I). This state transition frequency data is very useful for answering "what if" questions. Example 5.8 shows how we can determine the bandwidth requirement these workloads would place on the memory system.

EXAMPLE 5.8 Suppose that the integer-intensive applications run at a sustained 200 MIPS per processor and the floating-point-intensive applications at 200 MFLOPS per processor. Assuming that cache block transfers move 64 bytes on the data bus lines and that each bus transaction involves 6 bytes of command and address on the address lines, what is the traffic generated per processor?

Answer The first step is to calculate the amount of traffic per instruction. We determine what bus action is taken for each of the possible state transitions and therefore how much traffic is associated with each transaction. For example, an M → NP transition indicates that, due to a miss, a modified cache block needs to be written back. Similarly, an S → M transition indicates that an upgrade request must be issued on the bus. Flushing a modified block response to a bus transaction (e.g., the M → S or M → I transition) leads to a BusWB transaction as well. The bus transactions for all possible transitions are shown in Table 5.2. All transactions generate 6 bytes of address bus traffic and 64 bytes of data traffic, except BusUpgr, which only generates address traffic. We can now compute the traffic generated. Using Table 5.2, we can convert the state transitions per 1,000 memory references in Table 5.1 to bus transactions per 1,000 memory references and convert this to address and data traffic by multiplying by the traffic per transaction. Then, using the frequency of memory accesses in Table 4.1, we can convert this to traffic per instruction. Finally, multiplying by the assumed processing rate, we get the address and data bandwidth requirement for each application. The result of this calculation is shown by the leftmost bar for each application in Figure 5.18.[5] ■

5. For the Multiprog workload, to speed up the simulations, a 32-KB instruction cache is used as a filter before passing the instruction references to the 1-MB unified instruction and data cache. The state transition frequencies for the instruction references are computed based only on those references that missed in the L_1 instruction cache. This filtering does not affect how we compute data traffic, but it means that instruction traffic is computed differently. In addition, for Multiprog we present data separately for kernel instructions, kernel data references, user instructions, and user data references. A given reference may produce transitions of multiple types for user and kernel data. For example, if a kernel instruction miss causes a modified user data block to be written back, then we will have one transition for kernel instructions from NP → E/S and another transition for the user data reference category from M → NP.

Table 5.2 Bus Actions Corresponding to State Transitions in Illinois MESI Protocol

		To				
		NP	I	E	S	M
From	NP	—	—	BusRd	BusRd	BusRdX
	I	—	—	BusRd	BusRd	BusRdX
	E	—	—	—	—	—
	S	—	—	Not possible	—	BusUpgr
	M	BusWB	BusWB	Not possible	BusWB	—

The calculation in the preceding example gives the average bandwidth requirement under the assumption that the bus bandwidth is enough to allow the processors to execute at full speed. (In practice, bandwidth limitations may slow processors and events down, which in turn would lead to lower traffic per unit time.) This calculation provides a useful basis for sizing the number of processors that a system can support without saturating the bus. For example, on a machine such as the SGI Challenge with 1.2 GB/s of data bandwidth, the bus provides sufficient average bandwidth to support 16 processors on all the applications other than Radix for these problem sizes. A typical rule of thumb might be to leave 50% "headroom" to allow for burstiness of data transfers. If the Ocean and Multiprog workloads were also excluded, the bus could support up to 32 processors. If the bandwidth is not sufficient to support the application, the application will slow down. Thus, we would expect the speedup curve for Radix to flatten out quite quickly as the number of processors grows. In general, a multiprocessor is used for a variety of workloads, many with low per-processor bandwidth requirements, so the designer will choose to support configurations of a size that would overcommit the bus on the most demanding applications.

5.4.3 Impact of Protocol Optimizations

Given this base design point, we can evaluate protocol trade-offs under common machine parameter assumptions, as illustrated in Example 5.9.

EXAMPLE 5.9 We have described two invalidation protocols in this chapter—the basic three-state MSI protocol and the Illinois MESI protocol. The key difference is that the MESI protocol includes the existence of the exclusive state. How large is the bandwidth savings due to the E state?

Answer The main advantage of the E state is that no traffic need be generated when going from E → M. A three-state protocol would have to generate a BusUpgr transaction to acquire exclusive ownership for the memory block. To compute bandwidth savings, all we have to do is put a BusUpgr for the E → M transition in Table 5.2 and recompute the traffic as before. The middle bar in Figure 5.18 shows the resulting bandwidth requirements. ■

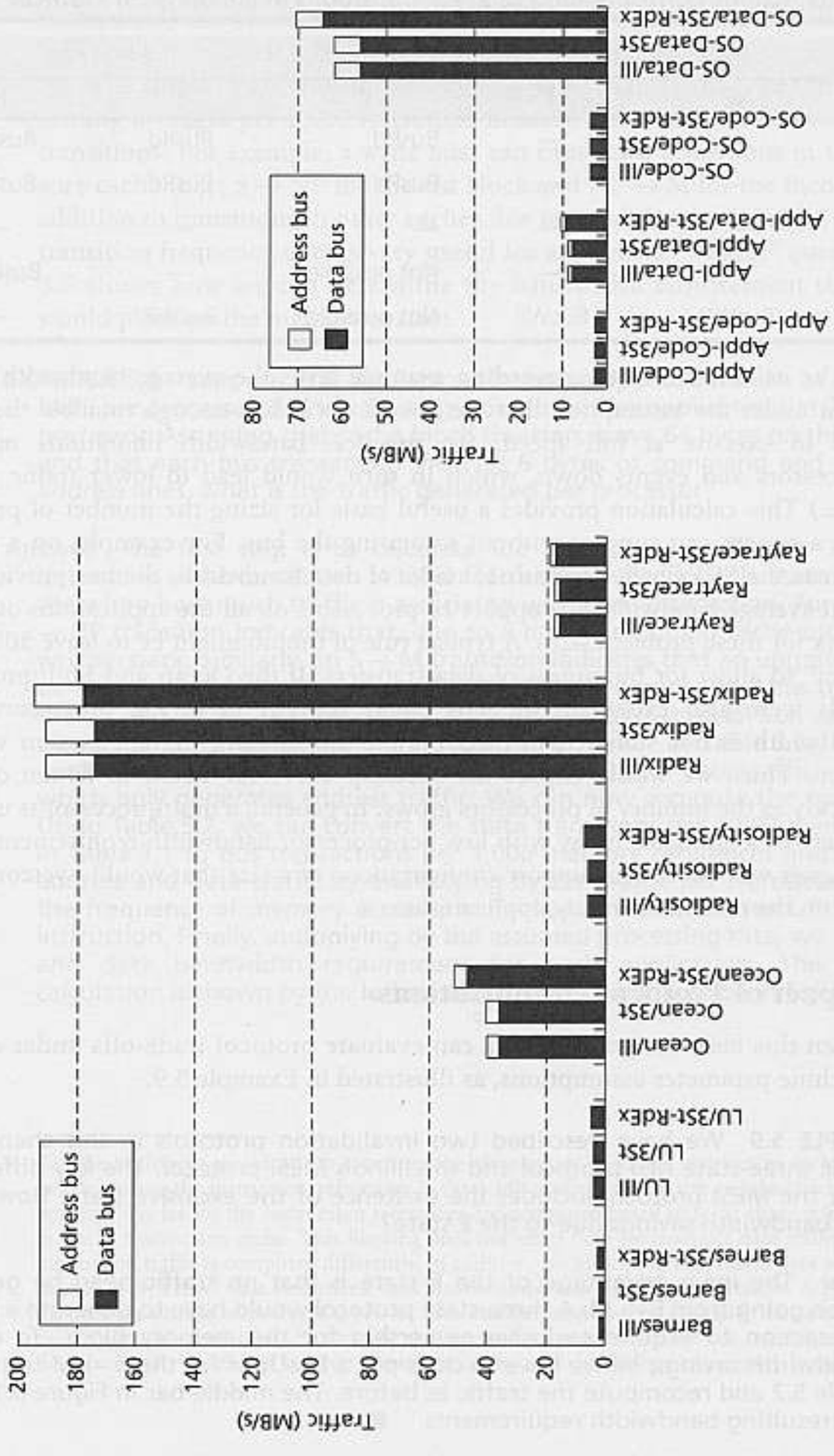

FIGURE 5.18 Per-processor bandwidth requirements for the various applications, assuming 200-MIPS/MFLOPS processors and 1-MB caches per processor. The left bar chart shows data for the parallel programs, and the right chart shows data for the Multiprog workload. The traffic is split into data traffic and address (including command) bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol (III), the middle bar for the case where we use the basic three-state invalidation protocol without the E state (3St), and the rightmost bar for the three-state protocol when we use BusRdX instead of BusUpgr for S → M transitions (3St-RdEx).

Example 5.9 illustrates how an intuitive rationale for a more complex design may not stand up to quantitative measurement of workloads. Contrary to expectations, the E state offers negligible savings in traffic. This is true even for the Multiprog workload, which consists primarily of sequential jobs and should have benefited most. The primary reason for this negligible gain is that the fraction of E → M transitions in Table 5.1 is quite small (i.e., blocks loaded in exclusive state by a read miss are not often written while still in that state). In addition, the BusUpgr transaction that would have been needed for the S → M transition in a three-state protocol takes only 6 bytes of address traffic and no data traffic. Example 5.10 examines the advantage of the BusUpgr transaction.

EXAMPLE 5.10 Recall that even in the three-state MSI protocol, a write that finds the memory block in shared state in the cache generates a BusUpgr request on the bus rather than a BusRdX. This saves bandwidth, as no data need be transferred for a BusUpgr, but it complicates the implementation, as we shall see. The question is, how much bandwidth are we saving for taking on the extra complexity?

Answer To compute the bandwidth for the less complex implementation and a three-state protocol, all we have to do is put in BusRdX in the E → M and S → M transitions in Table 5.2 (these would all be S → M transitions in the three-state MSI protocol) and then recompute the bandwidth numbers. The results for all applications are shown in the rightmost bar in Figure 5.18. While for most applications the difference in bandwidth is small, Ocean and Multiprog kernel data references show that it can be as large as 10–20% for some applications. ■

The performance impact of these differences in bandwidth requirement depends on how the bus transactions are actually implemented. However, this high-level analysis indicates where more detailed evaluation is required.

Finally, as discussed in Chapter 4, for the input data set sizes we are using it is important that we run the Ocean, Raytrace, and Radix applications for smaller cache sizes as well, to model the situation where an important working set does not fit in the cache hierarchy. We use 64-KB caches here, which fit all but the largest working set for these problem sizes. The raw state transition data for this case is presented in Table 5.3, and the per-processor bandwidth requirements are shown in Figure 5.19. As we can see, not having one of the critical working sets fit in the processor cache can dramatically increase the bus bandwidth required due to capacity misses. A 1.2-GB/s bus can now barely support 4 processors for Ocean and Radix and 16 processors for Raytrace.

5.4.4 Trade-Offs in Cache Block Size

The cache organization is a critical performance factor in all modern computers, but it is especially so in multiprocessors. In the uniprocessor context, cache misses are typically categorized into the "three Cs": compulsory, capacity, and conflict misses (Hill and Smith 1989; Hennessy and Patterson 1996). *Compulsory misses*, or *cold misses*, occur on the first reference to a memory block by a processor. *Capacity misses* occur when all the blocks that are referenced by a processor during the execution of a program do not fit in the cache (even with full associativity), so some

Table 5.3 State Transitions per 1,000 Memory References Issued by the Applications with Smaller Caches

Application			To				
			NP	I	E	S	M
Ocean	From	NP	0	0	26.2491	2.6030	15.1459
		I	1.3305	0	0	0.3012	0.0008
		E	21.1804	0.2976	452.580	0.4489	4.3216
		S	2.4632	1.3333	0	113.257	1.1112
		M	19.0240	0.0015	0	1.5543	387.780
Radix	From	NP	0	0	9.440787	2.557865	27.36084
		I	4.354862	0	0.00057	0.157565	1.499903
		E	8.148377	0.001329	140.9295	0.012339	0.126621
		S	3.825407	0.481427	0	102.4144	0.484464
		M	23.03084	5.629429	0	2.069604	717.1426
Raytrace	From	NP	0	0	7.2642	3.9742	0.1305
		I	0.0526	0	0.0003	0.2799	0.0000
		E	6.4119	0	131.944	0.7973	0.0496
		S	4.6768	0.3329	0	205.994	0.2835
		M	0.1812	0.0001	0	0.2837	660.753

The data assumes 16 processors, 64-KB four-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

blocks are replaced and later accessed again. *Conflict* or *collision misses* occur in caches with less than full associativity when the collection of blocks referenced by a program that maps to a single cache set does not fit in the set. They are misses that would not have occurred in a fully associative cache. Many studies have examined how cache size, associativity, and block size affect each category of miss.

Architecturally, capacity misses are reduced by enlarging the cache. Conflict misses are reduced by increasing the associativity or increasing the number of lines to map to in the cache (by increasing cache size or reducing block size). Cold misses can be reduced only by increasing the block size so that a single cold miss will bring in more data that may be accessed thereafter as well. What makes cache design challenging in uniprocessors is that these factors trade off against one another. For example, increasing the block size for a fixed cache capacity will reduce the number of blocks, so the reduced cold misses may come at the cost of increased conflict misses. Also, variations in cache organization can affect the miss penalty or the hit time and, therefore, perhaps the processor cycle time.

Cache-coherent multiprocessors introduce a fourth category of misses: *coherence misses*. These occur when blocks of data are shared among multiple caches. There

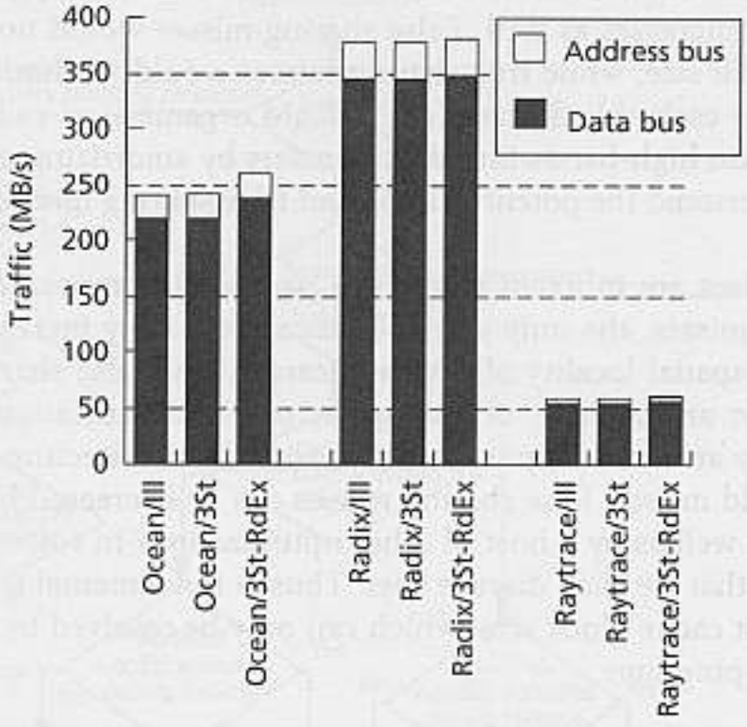

FIGURE 5.19 Per-processor bandwidth requirements for the various applications, assuming 200-MIPS/MFLOPS processors and 64-KB caches. The traffic is split into data traffic and address (including command) bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic three-state invalidation protocol without the E state (as described in Section 5.3.1), and the rightmost bar for the three-state protocol when we use BusRdX instead of BusUpgr for S → M transitions.

are two types: true sharing and false sharing misses. True sharing occurs when a data word produced (written) by one processor is used (read or written) by another. False sharing occurs when independent data words accessed by different processors happen to be placed in the same memory (cache) block, and at least one of the accesses is a write. The cache block size is not only the granularity (or unit) of the data fetched from the main memory, it is also typically used as the granularity of coherence. That is, on a write by a processor, the whole cache block is invalidated in other processors' caches, not just the word that is written.

More precisely, a *true sharing miss* occurs when one processor writes some words in a cache block, invalidating that block in another processor's cache, after which the second processor reads one of the modified words. It is called a "true" sharing miss because the miss truly communicates newly defined data values that are used by the second processor; such misses are essential to the correctness of the program, regardless of interactions with the machine organization or granularities. On the other hand, when one processor writes a word in a cache block and then another processor reads (or writes) a different word in the same cache block, the invalidation of the block and subsequent cache miss occurs as well, even though no useful values are being communicated between the processors. These misses are thus called *false sharing misses* (Dubois et al. 1993). As cache block size is increased, the probability of distinct variables being accessed by different processors but residing on the same

cache block increases. If at least some of these variables are written, the likelihood of false sharing misses increases as well. False sharing misses would not occur with a one-word cache block size, while true sharing misses would. Technology pushes in the direction of large cache block sizes (e.g., DRAM organization and access modes and the need to obtain high-bandwidth data transfers by amortizing overhead), so it is important to understand the potential impact of false sharing misses and how they may be avoided.

True sharing misses are inherent to a given parallel decomposition and assignment, so, like cold misses, the only way to reduce them is by increasing the block size and increasing spatial locality of communicated data. False sharing misses, on the other hand, are an example of the artifactual communication discussed in Chapter 3 since they are caused by interactions with the architecture. In contrast to true sharing and cold misses, false sharing misses can be decreased by reducing the cache block size, as well as by a host of other optimizations in software (orchestration) and hardware that we shall discuss later. Thus, a fundamental tension exists in determining the best cache block size, which can only be resolved by evaluating the options against real programs.

A Classification of Cache Misses

The flowchart in Figure 5.20 gives a detailed algorithm for classifying cache misses in cache-coherent multiprocessors.[6] Understanding the details is not critical for now—it is enough for the rest of the chapter to understand only the preceding definitions—but it adds insight and is a useful exercise. In the algorithm, the *lifetime* of a block in a cache is defined as the time interval during which the block remains valid in the cache, that is, the time from the occurrence of the miss that loads the block in the cache until its invalidation, replacement, or the end of the program. We cannot classify a cache miss when it occurs but only when the fetched memory block is replaced or invalidated in the cache, because it is only then that we know whether true sharing or only false sharing occurred during that lifetime. Let us consider the simple cases first. Cases 1 and 2 are straightforward cold misses occurring on previously unwritten blocks. Cases 7 and 8 reflect false and true sharing on a block that was previously invalidated in the cache but yet replaced by another. The type of sharing is determined by whether the specific word or words modified since the invalidation are actually used during the current lifetime. Case 9 is a straightforward capacity (or conflict) miss since the block was previously replaced from the cache and the words in the block have not been modified since last accessed. All of the other cases refer to misses that occur due to a combination of factors. For example, cases 4 and 5 are cold misses because this processor has never accessed the block before; however, some other processor had written the block, so there is also

6. In this classification, we do not distinguish conflict from capacity misses since both are a result of the available resources (set or entire cache) becoming full and the difference between them does not shed additional light on multiprocessor issues.

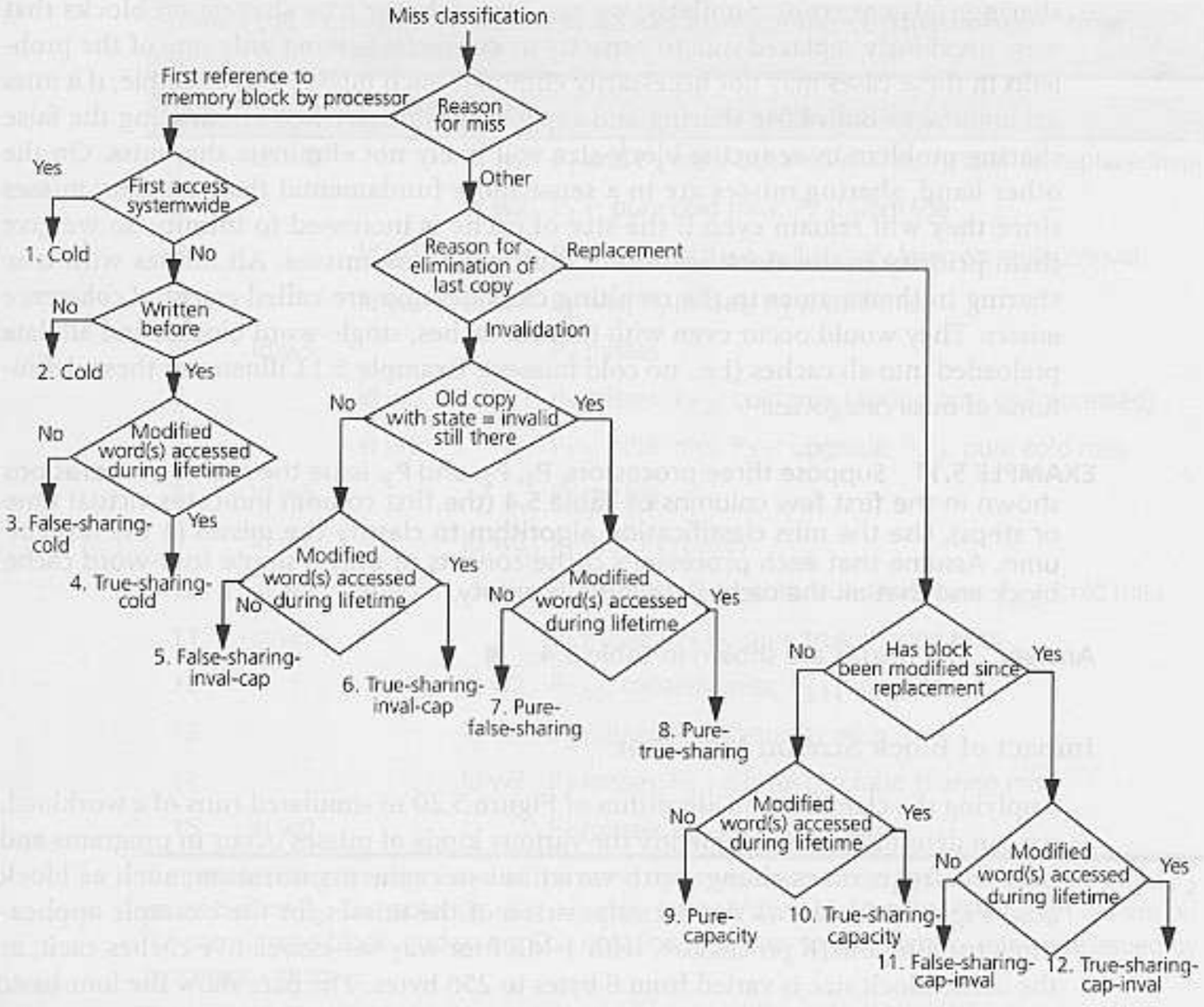

FIGURE 5.20 A classification of cache misses for shared memory multiprocessors. The four basic categories of cache misses in this classification are cold, capacity, true sharing, and false sharing misses (conflict misses are considered to be capacity misses for this purpose). Many mixed categories arise because there may be multiple causes for a miss. For example, a block may be first replaced from processor *A*'s cache, then written to by processor *B*, and then read back by processor *A*, making it a capacity-cum-invalidation false/true sharing miss. This would be labeled "false/true sharing cap-inval" in the classification since sharing takes priority and since the replacement happened before the invalidation (cases 11 and 12 in the figure). If the block were first invalidated in *A*'s cache, then the invalid block replaced, and then read again by *A*, it would be labeled "false/true sharing inval-cap" (cases 6 and 7). In terms of the four major categories, these misses all fall into true or false sharing misses, as appropriate. *Note:* the question "modified word(s) accessed during lifetime?" asks whether accesses are made by this processor in the current lifetime to word(s) within the cache block that have been modified since the last "essential coherence" miss to this block by this processor, where essential coherence misses correspond to categories 4, 6, 8, 10, and 12. This can only be determined when the current lifetime of the block ends.

sharing (false or true). Similarly, we can have false or true sharing on blocks that were previously replaced due to capacity or conflicts. Solving only one of the problems in these cases may not necessarily eliminate such misses. For example, if a miss occurs due to both false sharing and capacity problems, then eliminating the false sharing problem by reducing block size will likely not eliminate that miss. On the other hand, sharing misses are in a sense more fundamental than capacity misses since they will remain even if the size of cache is increased to infinity, so we give them priority in the classification of multiple-cause misses. All misses with true sharing in their names in the resulting classification are called *essential coherence misses*. They would occur even with infinite caches, single-word blocks, and all data preloaded into all caches (i.e., no cold misses). Example 5.11 illustrates these definitions of miss categories.

EXAMPLE 5.11 Suppose three processors, P_1, P_2, and P_3, issue the memory operations shown in the first few columns of Table 5.4 (the first column indicates virtual time or steps). Use the miss classification algorithm to classify the misses in the last column. Assume that each processor's cache consists of only a single four-word cache block and that all the caches are initially empty.

Answer The results are shown in Table 5.4. ■

Impact of Block Size on Miss Rate

Applying the classification algorithm of Figure 5.20 to simulated runs of a workload, we can determine how frequently the various kinds of misses occur in programs and how the frequencies change with variations in cache organization, such as block size. Figure 5.21 shows the decomposition of the misses for the example applications running on 16 processors, with 1-MB four-way set-associative caches each, as the cache block size is varied from 8 bytes to 256 bytes. The bars show the four basic types of misses: cold misses (cases 1 and 2), capacity—including conflict—misses (case 9), true sharing misses, (cases 4, 6, 8, 10, 12), and false sharing misses (cases 3, 5, 7, and 11). In addition, they show the frequency of *upgrades*—writes that find the block in the cache but in the shared state. Upgrades are different from the other types of misses since the cache already has the valid data and only needs exclusive ownership. While they are not included in the classification scheme of Figure 5.20, they are still usually considered to be misses since they generate traffic on the interconnect and can stall the processor.

For each individual application, the miss characteristics change with block size much as we would expect from our understanding of the program and the miss categories. Cold, capacity, and true sharing misses tend to decrease with increasing block size because the additional data brought in with each miss is accessed before the block is replaced, due to spatial locality. However, false sharing misses tend to increase with block size. In all cases, true sharing is a significant fraction of the misses, so even with ideal, infinite caches, the miss rate and bus bandwidth will not go to zero. However, the overall characteristics differ widely across programs. For example, the size of the true sharing component varies significantly. Some applica-

Table 5.4 Classifying Misses in an Example Reference Stream from Three Processors

Time	P_1	P_2	P_3	Miss Classification
1	ld w0		ld w2	P_1 and P_3 miss; but we will classify later on replace/inval
2			st w2	$P_{1.1}$: pure cold miss; $P_{3.2}$: upgrade
3		ld w1		P_2 misses, but we will classify later on replace/inval
4		ld w2	ld w7	P_2 hits; P_3 misses; $P_{3.1}$: cold miss
5	ld w5			P_1 misses
6		ld w6		P_2 misses; $P_{2.3}$: cold true sharing miss (w2 accessed)
7		st w6		$P_{1.5}$: cold miss; $P_{2.7}$: upgrade; $P_{3.4}$: pure cold miss
8	ld w5			P_1 misses
9	ld w6		ld w2	P_1 hits; P_3 misses
10	ld w2	ld w1		P_1, P_2 miss; $P_{1.8}$: pure true share miss; $P_{2.6}$: cold miss
11	st w5			P_1 misses; $P_{1.10}$: pure true sharing miss
12			st w2	$P_{2.10}$: capacity miss; $P_{3.11}$: upgrade
13			ld w7	P_3 misses; $P_{3.9}$: capacity miss
14			ld w2	P_3 misses; $P_{3.13}$: inval cap false sharing miss
15	ld w0			P_1 misses; $P_{1.11}$: capacity miss

If multiple references are listed in the same row, we assume that P_1 issues before P_2 and P_2 issues before P_3. The notation ld/st w*i* refers to load/store of word *i*. W1 through w4 are on the same cache block, and so on. The notation $P_{i,j}$ points to the memory reference issued by processor *i* at row *j*.

tions show a substantial increase in false sharing with block size, whereas others show almost none. Furthermore, the figure shows data only for the default data sets. In practice it is very important to examine the results as the input data set size and number of processors are scaled before drawing conclusions about the false sharing or spatial locality of an application (see Chapter 4). Let us investigate the properties of the applications that give rise to differences in miss characteristics observed at the machine level and that allow us to understand scaling qualitatively.

Relation to Application Structure

Multiword cache blocks exploit spatial locality by prefetching data surrounding the accessed address. Of course, beyond a point, larger cache blocks can hurt performance by (1) prefetching unneeded data, (2) causing increased conflict misses as the number of distinct blocks that can be stored in a finite cache decreases with increasing block size, and (3) causing increased false sharing misses. Spatial locality in parallel programs tends to be lower than in sequential programs because, when a

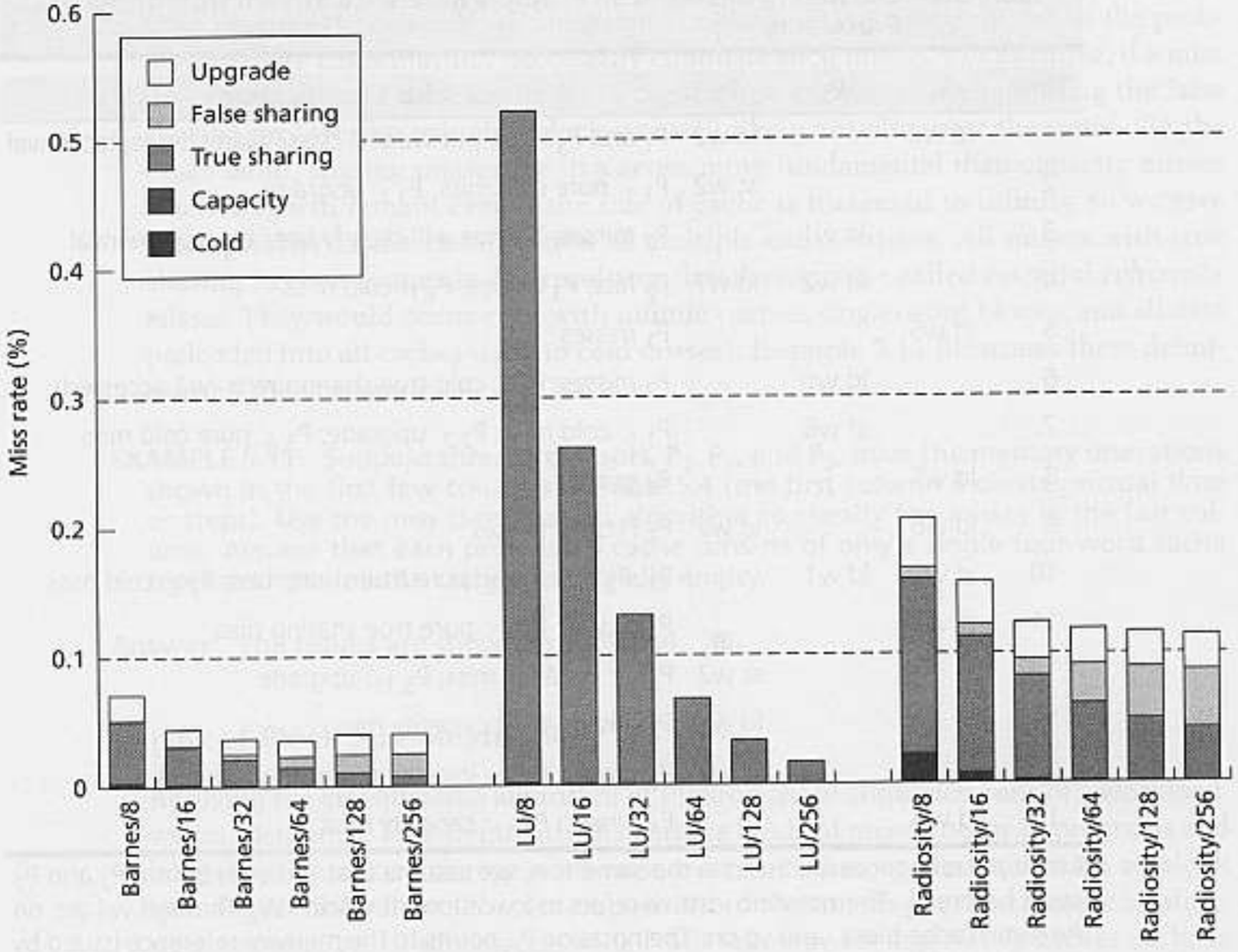

FIGURE 5.21(a) Breakdown of application miss rates as a function of cache block size for 1-MB caches per processor for Barnes-Hut, LU, and Radiosity applications. Conflict misses are included in capacity misses. The breakdown and behavior of misses vary greatly across applications, but we can observe some common trends. Cold misses and capacity misses tend to decrease quite quickly with block size as a result of spatial locality. True sharing misses also tend to decrease, whereas false sharing misses increase. While the false sharing component is usually small for small block sizes, it sometimes remains small and sometimes increases very quickly. Upgrades are shown at the top of the bars and without shading, so they can be ignored if desired.

memory block is brought into the cache, some of the data therein may belong to another processor and will not be used by the processor performing the miss. As an extreme example, some parallel programs assign adjacent elements of an array to different processors in order to ensure good load balance and in the process substantially decrease the spatial locality of the program.

The data in Figure 5.21 shows that LU and Ocean have good spatial locality and little false sharing even in the parallel case. The miss rates for many components drop proportionately to increases in cache block size, and false sharing misses are essentially nonexistent. This is in large part because these array-based codes use

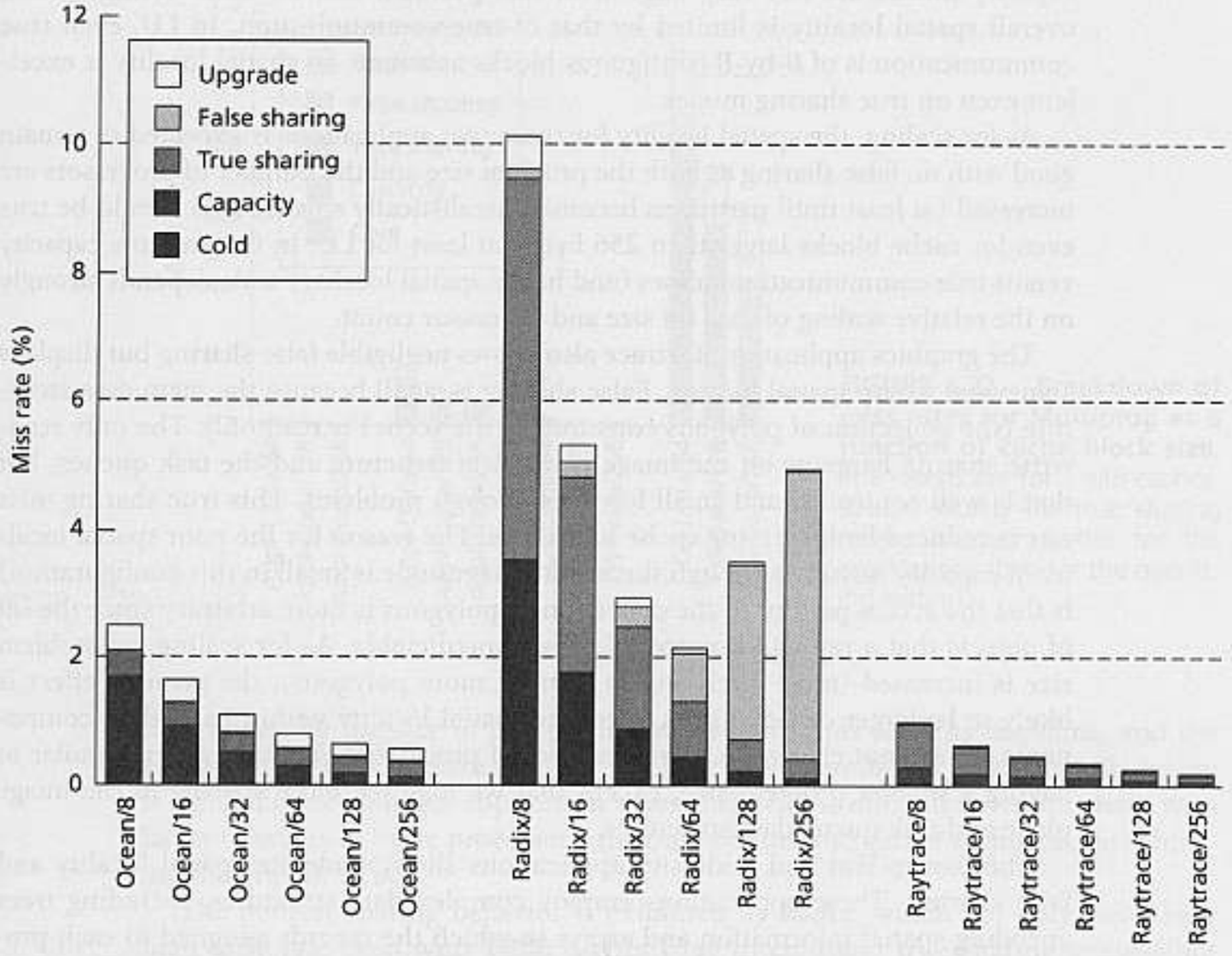

FIGURE 5.21(b) Breakdown of application miss rates as a function of cache block size for 1-MB caches per processor for Ocean, Radix, and Raytrace applications.

architecturally aware data structures, as discussed in Chapters 3 and 4. For example, a grid in Ocean is not represented as a single 2D array (which can introduce substantial false sharing at column-oriented partition boundaries) but as a 4D array: a 2D array of blocks, each of which is itself a 2D array. Such structuring, by programmers or compilers, ensures that most accesses are unit stride and over substantial, contiguous blocks of data, thus the nice behavior.

In Ocean, capacity misses are significant, but they are to the interior elements of a process's partition, so they have very good spatial locality. One difference with LU is that true sharing misses in Ocean do not exhibit such good spatial locality. Most of the true sharing misses are to elements at the borders of neighboring partitions. These exhibit good spatial locality at row-oriented borders where the data to be fetched is contiguous in the address space. However, when a processor accesses an element at a column-oriented border, it fetches an entire cache block of interior elements of its neighbor's partition, which it will not use and therefore wastes. Since

capacity misses are not very large with this problem and machine configuration, overall spatial locality is limited by that of true communication. In LU, even true communication is of B-by-B contiguous blocks at a time, so spatial locality is excellent even on true sharing misses.

As for scaling, the spatial locality for these two applications is expected to remain good with no false sharing as both the problem size and the number of processors are increased (at least until partitions become unrealistically small). This should be true even for cache blocks larger than 256 bytes, at least for LU. In Ocean, how capacity versus true communication misses (and hence spatial locality) scale depends strongly on the relative scaling of data set size and processor count.

The graphics application Raytrace also shows negligible false sharing but displays somewhat worse spatial locality. False sharing is small because the main data structure (the collection of polygons constituting the scene) is read-only. The only read-write sharing happens on the image plane data structure and the task queues, but that is well controlled and small for large enough problems. This true sharing miss rate is reduced by increasing cache block size. The reason for the poor spatial locality of capacity misses (although the overall magnitude is small in this configuration) is that the access pattern to the collection of polygons is quite arbitrary since the set of objects that a ray will bounce off of is unpredictable. As for scaling, as problem size is increased (most likely in the form of more polygons), the primary effect is likely to be larger capacity miss rates; the spatial locality within individual components should not change. A larger number of processors is in many ways similar to having a smaller problem size, except that we may see more sharing in the image plane and task queue data structures.

The Barnes-Hut and Radiosity applications show moderate spatial locality and false sharing. These applications employ complex data structures, including trees encoding spatial information and arrays in which the records assigned to each processor are not contiguous in memory. For example, Barnes-Hut operates on particle records stored in an array. As the application proceeds and particles move in physical space, particle records get reassigned to different processors, with the result that after some time adjacent particles in the array most likely belong to different processors. Spatial locality is exploited well within a particle record but not very well across records. False sharing becomes a problem at large block sizes for different reasons. First, different processors may write to different records that share a cache block. Second, a particle data structure (record) contains both fields that are being modified by the owner of that particle in a phase (e.g., the current force on this particle in the force calculation phase) and fields that are read by other processors and are not being modified in this phase (e.g., the current position of the particle). Since these two fields may fall in the same cache block for large block sizes, false sharing results. It is possible to eliminate such false sharing by splitting the particle data structure according to the access patterns of the fields, but that is not done in this program since the absolute magnitude of the miss rate is small. As problem size and the number of processors are scaled, the miss rate behavior of Barnes-Hut is expected to change little. This is because the working set size changes very slowly (as the log of the number of particles, unlike Ocean and Raytrace), spatial locality is

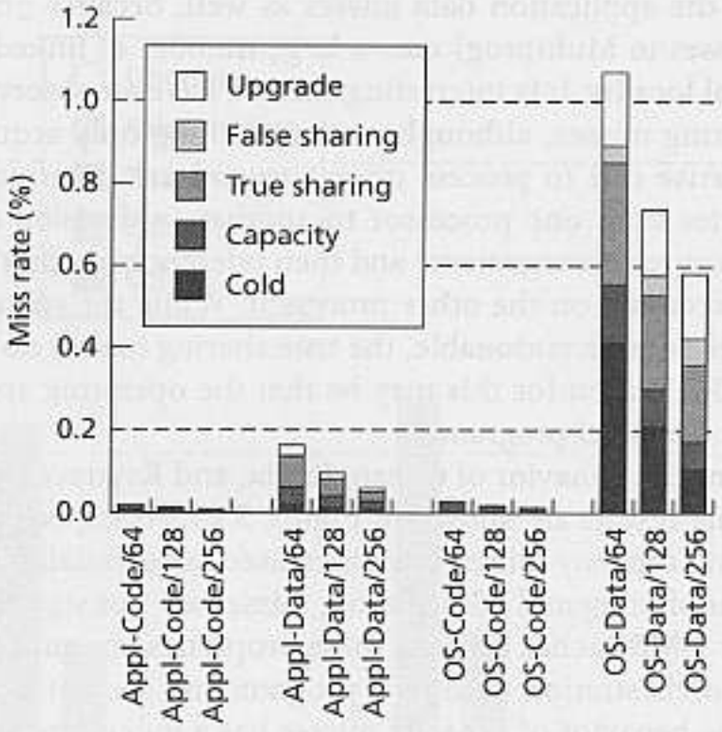

FIGURE 5.22 Breakdown of miss rates for Multiprog as a function of cache block size. The results are for 1-MB caches. Spatial locality for true sharing misses is much better for the applications than for the operating system.

determined by the size of one particle record and thus remains the same, and the sources of false sharing are not very sensitive to the number of processors. Radiosity is a much more complex application whose behavior is difficult to reason about with larger data sets or more processors; the only option is to gather empirical data showing the growth trends.

The poorest sharing behavior is exhibited by Radix, which not only has a very high miss rate even with 1-MB caches (due to cold and true sharing misses) but which gets significantly worse due to false sharing misses for block sizes of 128 bytes or more. The effect of false sharing in Radix was illustrated in Chapter 4. Let us now examine how it is governed. Consider sorting 256-K keys, using a radix of 1,024 and 16 processors. On average, this results in 16 keys per radix per processor (64 bytes of data), which are then written to a contiguous portion of a global array at an unpredictable starting point. Adjacent 64-byte chunks in this array are written by different processors. If the cache block size is larger than 64 bytes, the high potential for false sharing is clear. As the problem size is increased we will clearly see much less false sharing. The effect of increasing the number of processors is exactly the opposite. Radix illustrates quite dramatically that it is not sufficient to look at a given problem size and number of processors and, based on that, draw conclusions of whether or not false sharing or spatial locality is a problem. It is very important to understand how the results are dependent on the key parameters chosen in the experiment and how these parameters may vary in reality.

Data for the Multiprog workload for 1-MB caches is shown in Figure 5.22. The data is shown separately for user code, user data, kernel code, and kernel data. For code, there are only cold and capacity misses. Furthermore, we see that the spatial locality in operating system data references is not very good. This is true, to a some-

what lesser extent, for the application data misses as well, because gcc (the main application causing misses in Multiprog) uses a large number of linked lists, which do not offer good spatial locality. It is interesting that we have an observable fraction of application true sharing misses, although we are running only sequential applications. These misses arise due to process migration and are incurred when a sequential process migrates from one processor to another (a decision made by the operating system for resource management) and then references memory blocks that it wrote while it was executing on the other processor. While the spatial locality in cold and capacity misses is quite reasonable, the true sharing misses do not decrease at all for kernel data. One reason for this may be that the operating system has not been well structured as a parallel program.

Finally, let us examine the behavior of Ocean, Radix, and Raytrace for smaller 64-KB caches. The miss rate results are shown in Figure 5.23. As expected, the overall miss rates are higher, and capacity misses have increased substantially. The effects of cache block size for true sharing and false sharing misses are not significantly different from the results for 1-MB caches because these properties are quite fundamental to the assignment and orchestration used by a program and are not too sensitive to cache size. However, the behavior of capacity misses has a much larger effect on the behavior of the overall miss rate. For example, in Ocean, capacity misses now dominate sharing misses; since they have much better spatial locality, the overall miss rate decreases much more quickly with increasing block size than it did with 1-MB caches. (Very large blocks in a small cache can have the problem that blocks may be replaced from the cache due to conflicts before the processor has had a chance to reference all of the words in them.) In Raytrace, capacity misses have somewhat worse spatial locality than true sharing misses, so the overall benefits of large blocks look worse with smaller caches. Results for false sharing and spatial locality for other applications can be found in the literature (Torrellas, Lam, and Hennessy 1994; Jeremiassen and Eggers 1991; Woo et al. 1995).

While larger cache blocks reduce the miss rate for most of our applications, within the range of block sizes we consider they have two important potential disadvantages. First, they can increase the cost of each miss since more data has to be transferred across the bus (although techniques like only waiting for the referenced word to arrive before allowing the processor to proceed, called a *critical word restart* approach, can alleviate this). Second, they increase traffic, and hence contention, if the whole block is not useful.

Impact of Block Size on Bus Traffic

Let us briefly examine the impact of cache block size on bus traffic rather than miss rate. While the number of misses and total traffic generated are clearly related, their impact on observed performance can be quite different. Misses have a cost that may contribute directly to performance, even though modern microprocessors try hard to hide the latency of misses by overlapping it with other activities. Traffic, on the

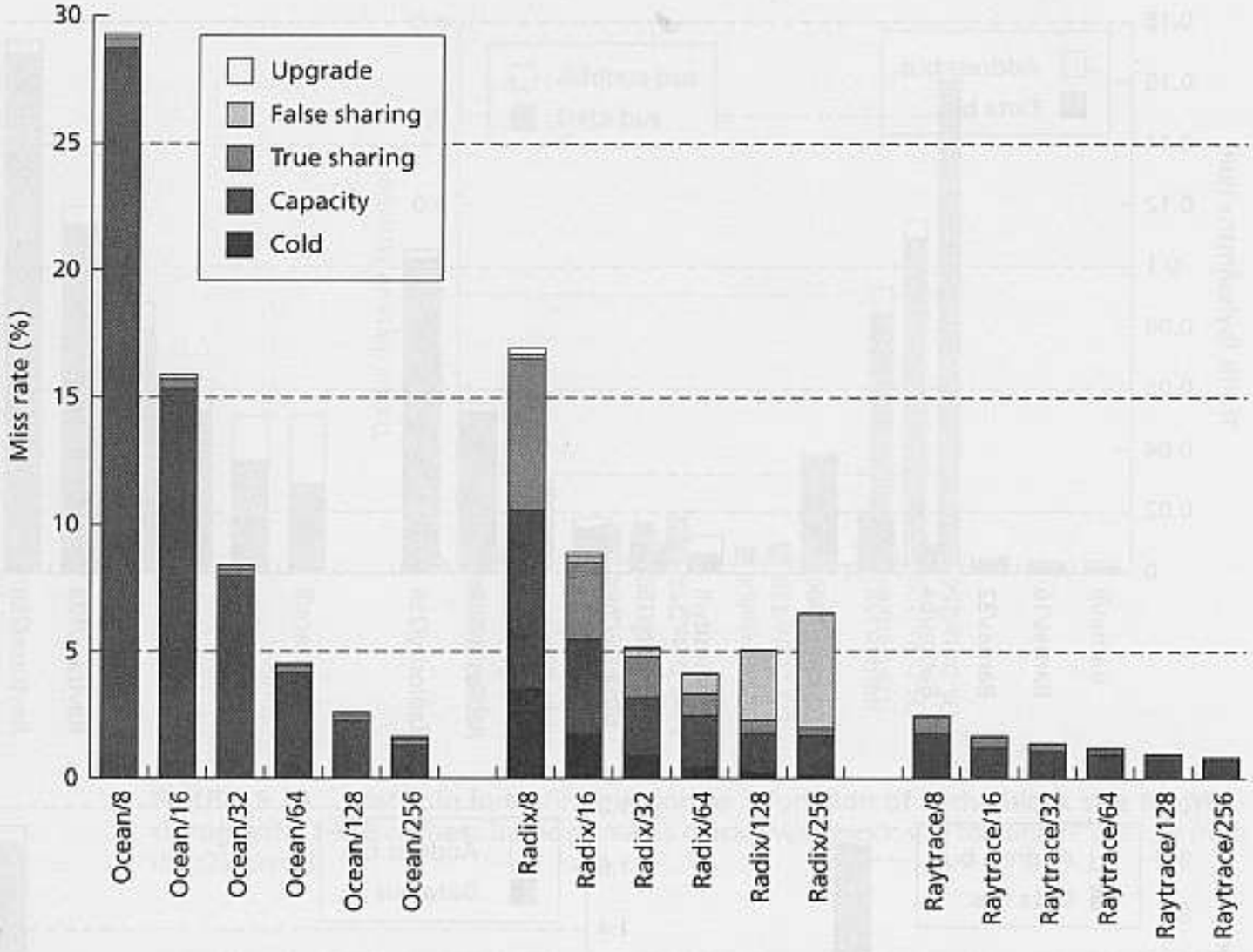

FIGURE 5.23 Breakdown of application miss rates as a function of cache block size for 64-KB caches. Capacity misses are now a much larger fraction of the overall miss rate. Capacity miss rates decrease differently with block size for different applications.

other hand, affects performance indirectly by causing contention and hence increasing the cost of other misses. For example, if an application program's misses are reduced significantly by increasing the cache block size, but the bus traffic is increased by 50%, this might be a reasonable trade-off if the application was originally using only 10% of the available bus and memory bandwidth. Increasing the bus and memory utilization to 15% is unlikely to increase the miss latencies significantly. However, if the application was originally using 75% of the bus and memory bandwidth, then increasing the block size is probably a bad idea.

Figure 5.24 shows the total bus traffic for our applications in bytes/instruction or bytes/FLOP as the cache block size is varied. Three key points can be observed from this graph. First, traffic behaves very differently than miss rate. Only LU shows monotonically decreasing total traffic for the block sizes used. Most other applications see a doubling or tripling of traffic as block size becomes large. Second, the

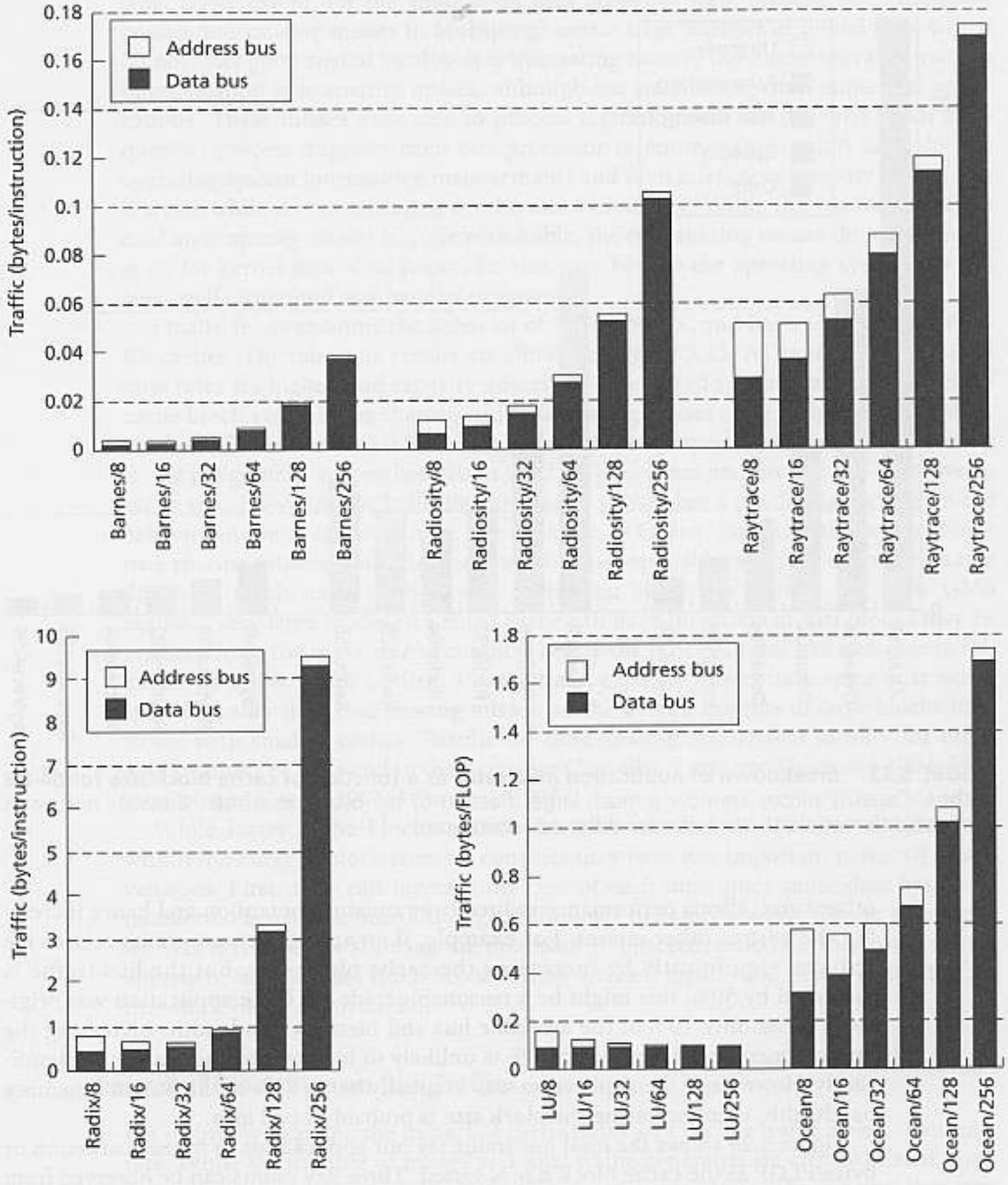

FIGURE 5.24 Traffic (in bytes/instruction or bytes/FLOP) as a function of cache block size with 1-MB caches per processor. Data traffic increases quite quickly with block size when communication misses dominate, except for applications like LU that have excellent spatial locality on all types of misses. Address (including command) bus traffic tends to decrease with block size since the miss rate and, hence, number of blocks transferred decrease.

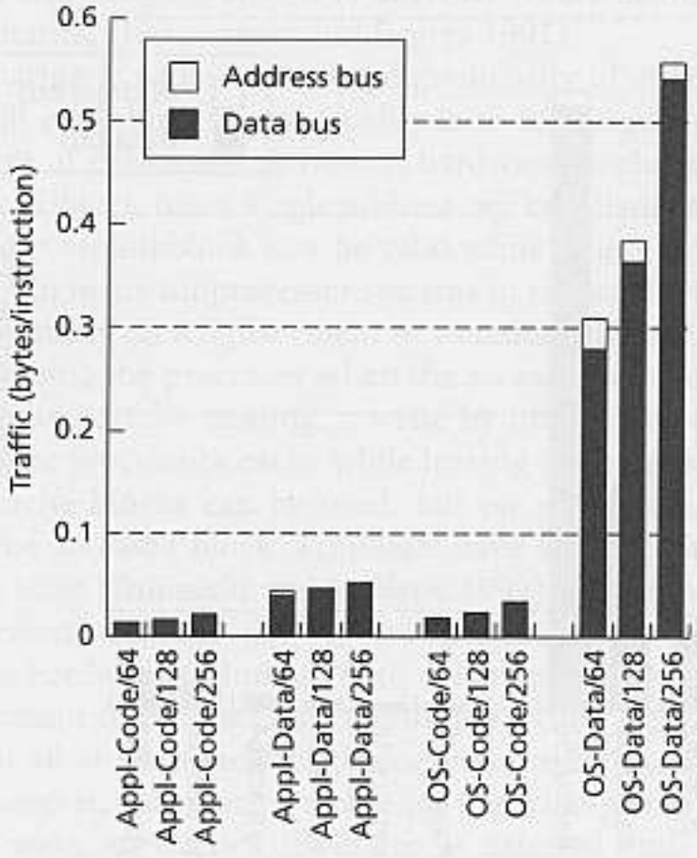

FIGURE 5.25 Traffic in bytes/instruction as a function of cache block size for Multiprog with 1-MB caches. Traffic increases quickly with block size for data references from the OS kernel.

overall traffic requirements for the applications are still small, even for 256-byte block sizes, with the exception of Radix. Radix's large bandwidth requirements (approximately 650 MB/s per processor for 128-byte cache blocks, assuming a sustained 200-MIPS processor) reflect its false sharing problems at large block sizes. Third, the constant address and command traffic overhead for each bus transaction or miss comprises a significant fraction of total traffic for small block sizes. Hence, although actual application data traffic usually increases as we increase the block size due to poor spatial locality, the total traffic is often minimized at 16–32 bytes rather than 8 bytes due to the amortization of the overhead with improved miss rates.

Figure 5.25 shows the traffic data for Multiprog. While the increase in traffic from 64-byte cache blocks to 128-byte blocks is small, the jump at 256-byte blocks is much more substantial (primarily due to kernel data references). Finally, Figure 5.26 shows the traffic results for 64-KB caches for the three relevant applications. For Ocean, even 64- and 128-byte cache blocks don't look so bad, due to the dominance of capacity misses that have good spatial locality.

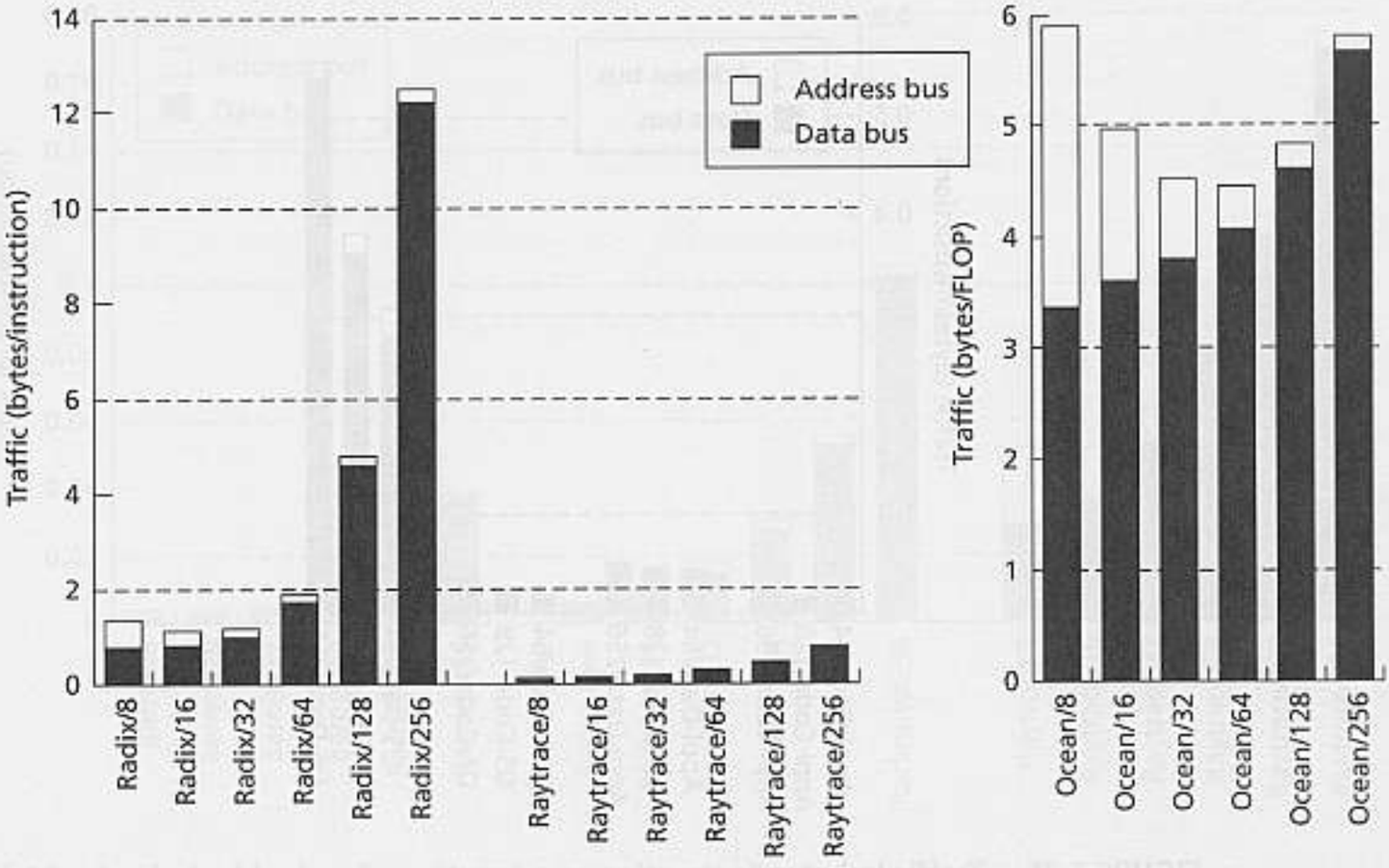

FIGURE 5.26 Traffic (in bytes/instruction or bytes/FLOP) as a function of cache block size with 64-KB caches per processor. Traffic increases more slowly now for Ocean than with 1-MB caches since the capacity misses that now dominate exhibit excellent spatial locality (traversal of a process's assigned subgrid). However, traffic in Radix increases quickly once the threshold block size that causes false sharing is exceeded.

Alleviating the Drawbacks of Large Cache Blocks

The trend toward larger cache block sizes is driven by the increasing gap between processor performance and memory access time. The larger block size amortizes the cost of the bus transaction and memory access across a greater amount of data. The increasing density of processor and memory chips makes it possible to employ large first-level and second-level caches so that the prefetching of data obtained through a larger block size dominates the small increase in conflict misses. However, this trend may bode poorly for multiprocessor designs because false sharing becomes a larger problem. Fortunately, hardware and software mechanisms can be employed to counter the effects of large block size.

Software techniques to reduce false sharing and improve locality on coherence misses are discussed in detail later in the chapter. They essentially involve organizing data structures or work assignments so that data accessed by different processes is not interleaved finely in the shared address space. One example is the use of higher-dimensional arrays so blocks or partitions are wholly contiguous. Compiler

techniques have also been developed to automate some methods of laying out data to reduce false sharing (Jeremiassen and Eggers 1991).

Since false sharing is caused by a large granularity of coherence, the way to reduce it while still exploiting spatial locality is to use large blocks for data transfer but a smaller unit of coherence. A natural hardware mechanism is the use of subblocks. Each cache block has a single address tag but distinct state bits for each of several subblocks. One subblock may be valid while others are invalid or dirty. This technique is used in many uniprocessor systems to reduce the amount of data that is copied back to memory on a replacement or to reduce the memory access time on a read miss by resuming the processor when the accessed subblock is present (critical word restart). To avoid false sharing, a write by one processor may invalidate the subblock in another processor's cache while leaving the other subblocks valid. Alternatively, small cache blocks can be used, but on a miss the system can prefetch blocks beyond the accessed block. Proposals have also been made for caches with adjustable block sizes (Dubnicki and LeBlanc 1992). The disadvantage of these approaches is increased state and complexity beyond a commodity cache design.

A more subtle hardware technique is to delay propagating or applying invalidations from a processor until it has issued multiple writes. Delaying invalidations and performing them all at once reduces the occurrence of intervening read misses to those blocks. However, this sort of technique can change the memory consistency model in subtle ways, so further discussion is deferred until Chapter 9 where we consider weaker consistency models in the context of scalable machines. Another hardware technique to reduce false sharing is the use of update- rather than invalidation-based protocols.

5.4.5 Update-Based versus Invalidation-Based Protocols

Whether writes should cause other cached copies to be updated or invalidated has been the subject of considerable debate. Various vendors have taken different stands and, in fact, have changed their position from one design to the next. The controversy arises because the relative performance of update-based versus invalidation-based protocols depends strongly on the sharing patterns exhibited by the workload and on the cost of various underlying operations. Intuitively, if the processors that were using the data before it was updated (written) are likely to want to see the new values in the future, updates should perform better than invalidations. However, if the processors holding the old data are never going to use it again, the update traffic is useless and just consumes interconnect and controller resources. Invalidations would clean out the old copies and eliminate the apparent sharing. This "pack rat" phenomenon with update protocols is especially irritating under multiprogrammed use of a machine, when sequential processes migrate from processor to processor under OS control so that useless updates are performed in caches of processors that are no longer running that process. It is easy to construct cases in which either scheme does substantially better than the other, as illustrated by Example 5.12.

EXAMPLE 5.12 Consider the following two program reference patterns:

- *Pattern 1:* Repeat k times; processor 1 writes a new value into variable V and processors 2 through P read the value of V. This represents a one-producer-many-consumer scenario that may arise, for example, when processors are accessing a highly contended flag for one-to-many event synchronization.
- *Pattern 2:* Repeat k times; processor 1 writes M times to variable V and then processor 2 reads the value of V. This represents a sharing pattern that may occur between pairs of processors, where the first successfully computes and accumulates values into a variable and then when the accumulation is complete, another processor reads the value.

What is the relative cost for update- and invalidation-based protocols in terms of the number of cache misses and bus traffic? Assume that an invalidation/upgrade transaction consumes 6 bytes (5 bytes for address plus 1 byte for command), an update takes 14 bytes (6 bytes for address and command and 8 bytes of data for the updated word), and a regular cache miss takes 70 bytes (6 bytes for address and command plus 64 bytes of data corresponding to cache block size). Also assume that $P = 16$, $M = 10$, $k = 10$, and that all caches initially are empty.

Answer With an update scheme in pattern 1, the first iteration on all P processors will incur a regular cache miss (including processor 1 when it writes) plus an update due to the write. In subsequent $k - 1$ iterations, no more misses will occur and only one update per iteration will be generated. Thus, overall we will see misses = P = 16; traffic = $P \times$ RdMiss + $(k - 1) \times$ Update = $16 \times 70 + 10 \times 14 = 1{,}260$ bytes.

With an invalidate scheme, all P processors will incur a regular cache miss in the first iteration. In subsequent $k - 1$ iterations, processor 1 will generate an upgrade, but all others will experience a read miss. Thus, counting upgrades as misses, overall we will see misses = $P + (k - 1) \times P = 16 + 9 \times 16 = 160$, of which 151 are read misses and 9 are upgrades; traffic = read misses $\times$ RdMiss + $(k - 1) \times$ Upgrade = $151 \times 70 + 9 \times 6 = 10{,}624$ bytes.

With an update scheme on pattern 2, the first iteration will incur two regular cache misses, one for processor 1 and the other for processor 2. In subsequent $k - 1$ iterations, no more misses will be generated, but M updates will be generated in each iteration. Thus, overall we will see misses = 2; traffic = $2 \times$ RdMiss + $M \times (k - 1) \times$ Update = $2 \times 70 + 10 \times 9 \times 14 = 1{,}400$ bytes.

With an invalidate scheme, two regular cache misses will occur in the first iteration. In subsequent $k - 1$ iterations, one upgrade (for the first write only) plus one regular read miss will be generated in each iteration. Thus, counting upgrades as misses, overall we will see misses = $2 + (k - 1) \times 2 = 2 + 9 = 11$; traffic = misses $\times$ RdMiss + $(k - 1) \times$ Upgrade = $11 \times 70 + 9 \times 6 = 824$ bytes. ■

These example patterns suggest that it might be possible to design schemes that capture the advantages of both update and invalidate protocols. The success of such schemes will depend on their costs and on the sharing patterns for real parallel programs and workloads. Let us briefly explore the design options and then employ workload-driven evaluation.

Combining Update- and Invalidation-Based Protocols

One way to take advantage of both update and invalidate protocols is to support both in hardware and to decide dynamically at page granularity whether coherence

for a given page is to be maintained using an update or an invalidate protocol. The decision about the choice of protocol can be indicated by making a system call. The main advantage of such schemes is that they are relatively easy to support; they utilize the TLB to indicate to the rest of the coherence subsystem which of the two protocols to use. The main disadvantage of such schemes is the burden they put on the programmer to choose protocols for pages or data structures. The decision task is also made difficult because of the coarse granularity at which control is made available; data structures that desire different protocols may fall on the same page.

An alternative is to choose the protocol at a cache block granularity, by observing the sharing behavior at run time. Ideally, for each write, we would like to be able to peer into the future references that will be made to that cache block by all processors and then decide whether to invalidate other copies or to do an update. Since this information is obviously not available, and since there are substantial perturbations due to cache replacements and false sharing, a more practical scheme is needed.

So-called competitive schemes change the protocol for a block between invalidate and update in hardware based on observed patterns at run time. The key attribute of such schemes is that if a wrong decision is made once for a cache block, the losses due to that wrong decision should be kept bounded and small (Karlin et al. 1986). For instance, if a block is currently using update mode, it should not remain in that mode if one processor is continuously writing to it but none of the other processors are reading values from it.

One class of schemes that has been proposed to bound the losses of update protocols works as follows (Grahn, Stenstrom, and Dubois 1995). Starting with the base Dragon update protocol described in Section 5.3.3, associate a countdown counter with each block. Whenever a cache block is accessed by the local processor, the counter value for that block is reset to a threshold value k. Every time an update is received for a block, the counter is decremented. If the counter goes to zero, the block is locally invalidated. The consequence of the local invalidations is that the next time an update is generated on the bus, it may find that no other cache has a valid copy; in that case, that block will switch to the modified state (as per the Dragon protocol) and will stop generating updates. If some other processor now accesses that block, the block will again switch to shared state and this mixed protocol will again start generating updates.

A related approach implemented in the Sun SparcCenter 2000 is to selectively invalidate rather than update with some probability that is a parameter set when configuring the machine (Catanzaro 1997). Other mixed approaches may also be used. For example, one approach uses an invalidation-based protocol for first-level caches and, by default, an update-based protocol for second-level caches. However, if the L_2 cache receives a second update for the block while the block in the L_1 cache is still invalid, then the block is invalidated in the L_2 cache as well. When the block is thus invalidated in all other L_2 caches, writes to the block no longer cause updates.

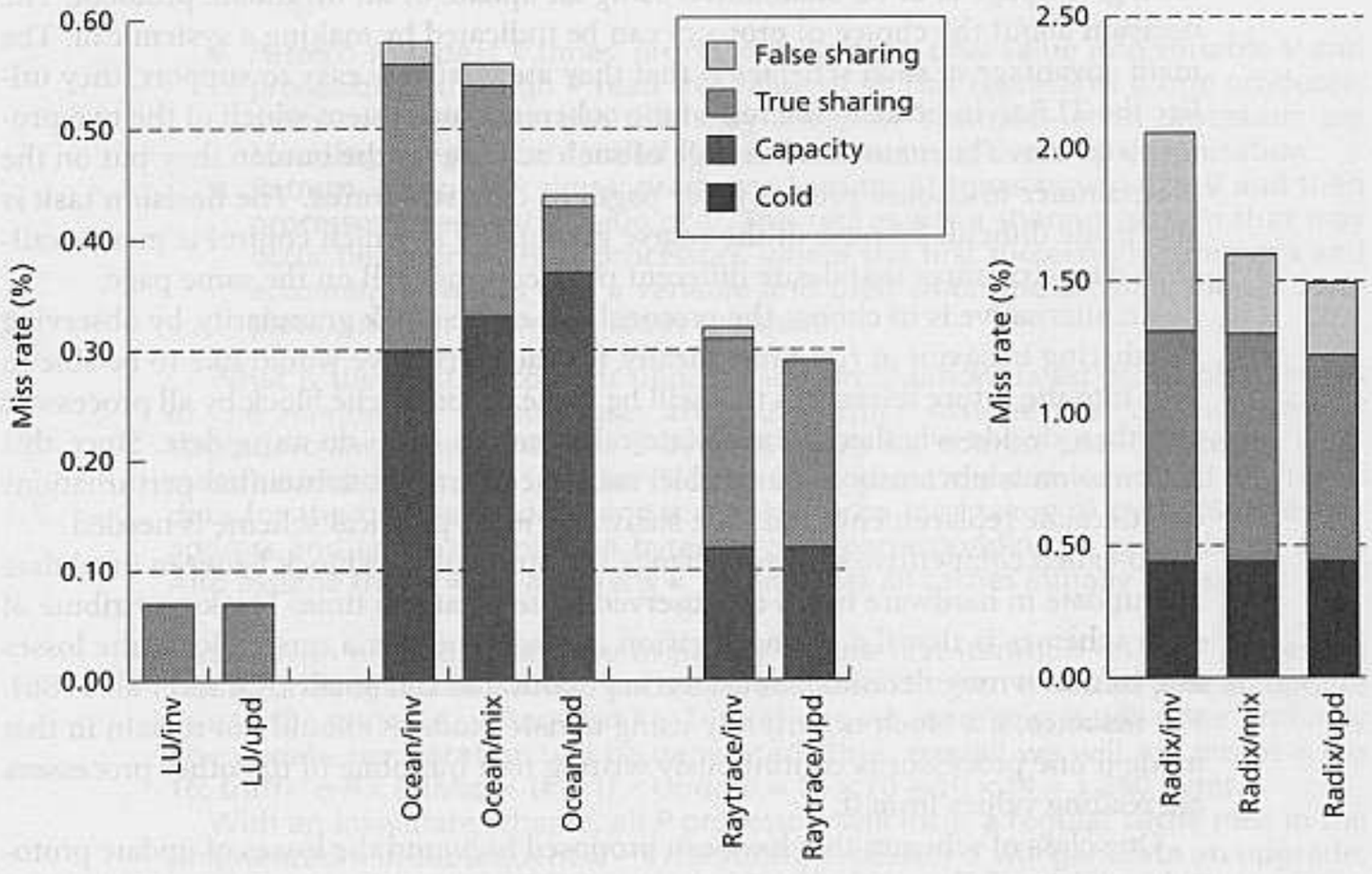

FIGURE 5.27 Miss rates and their decomposition for invalidate, update, and hybrid protocols. The data assumes 1-MB caches, 64-byte cache blocks, four-way set associativity, and threshold $k = 4$ for hybrid protocol.

Workload-Driven Evaluation

To assess the trade-offs among invalidate, update, and the mixed protocols just described, Figure 5.27 shows the miss rates by category for four applications using the default 1-MB four-way set-associative caches with a 64-byte block size. The mixed protocol used is the threshold-based scheme just described. We see that for applications with significant capacity miss rates, the misses sometimes increase with an update protocol. This makes sense because the protocol (with LRU replacement in a set) keeps data in processor caches that would have been removed by an invalidation protocol. For applications with significant true sharing or false sharing miss rates, these categories decrease with an update protocol: after a write update, the other caches holding the blocks can access them without a miss. Overall, the update protocol appears to be advantageous for the sum of these three categories and the mixed protocol falls in between. The category that is not shown in this figure, however, is the upgrade or update operations for these protocols. This data is presented in Figure 5.28. Note that the scale of the graphs has changed because update operations are roughly four times more prevalent than misses. It is useful to separate these operations from other misses because the way they are handled in the machine is

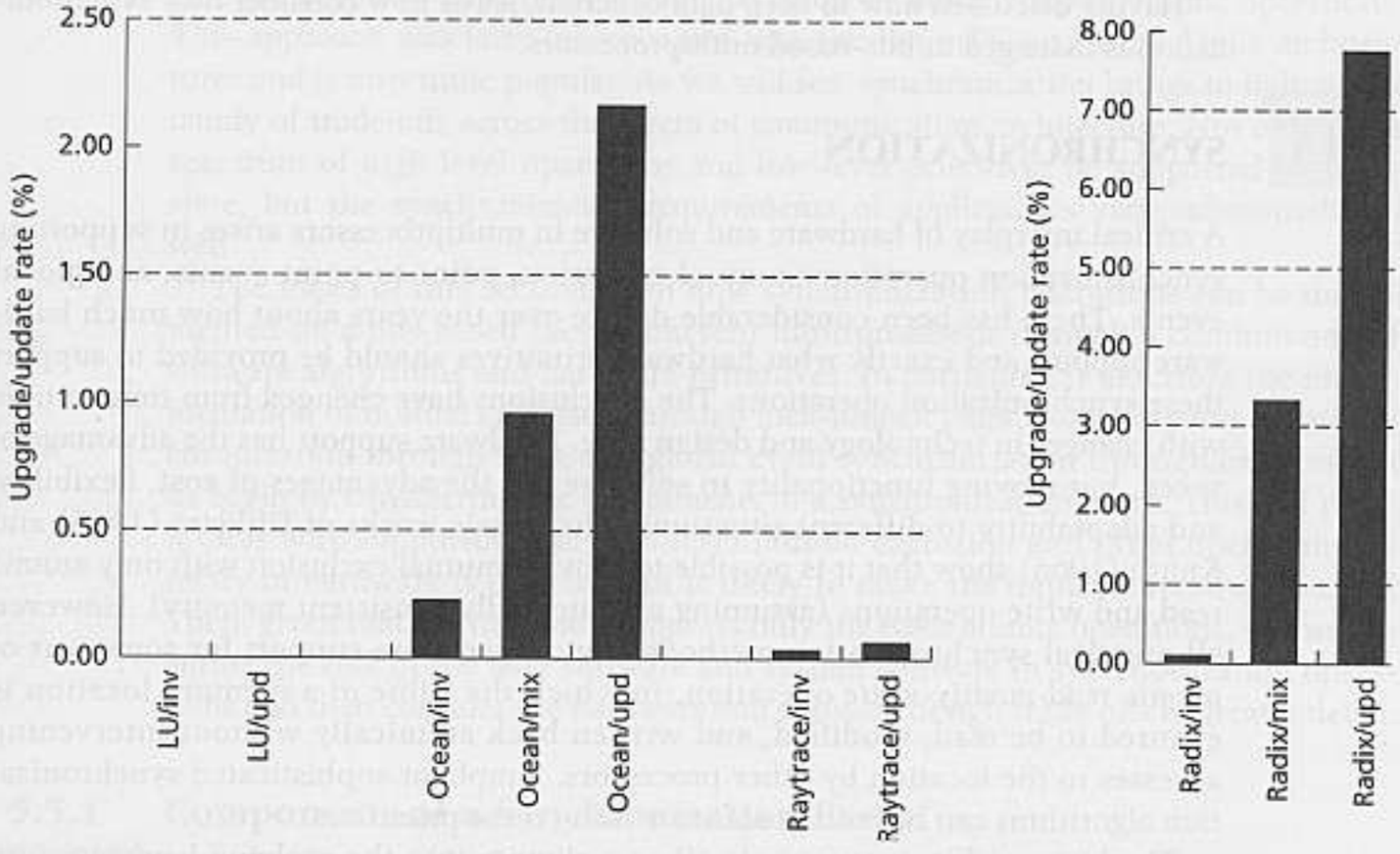

FIGURE 5.28 Upgrade and update rates for invalidate, update, and mixed protocols. The data assumes 1-MB caches, 64-byte cache blocks, four-way set associativity, and threshold $k = 4$ for hybrid protocol. Rates are measured relative to total memory references.

likely to be different. Updates are a single-word write rather than a full cache block transfer. Because the data is being pushed from where it is being produced, it may arrive at the consumer before it is needed. Even for the producer, the latency of update and upgrade operations may be less critical than that of misses since it is quite easily hidden from the processor's critical path (see Chapter 11).

Unfortunately, the traffic associated with updates is quite substantial. In large part, this occurs because multiple writes are made by a processor to the same block before a read, all generating updates. With the invalidate protocol, the first of these writes may cause an invalidation, but the rest can simply accumulate locally in the block and be transferred in one bus transaction on a flush or a write back (see Example 5.12). The increased traffic causes contention and can greatly increase the cost of misses. Sophisticated update schemes might attempt to delay the update to achieve a similar effect (by merging writes in the write buffer) or use other techniques to reduce traffic and improve performance (Dahlgren 1995). However, the increased bandwidth demand, the complexity of supporting updates, the trend toward larger cache blocks, and the pack rat phenomenon with the important case of multiprogrammed sequential workloads underlie the trend away from update-based protocols in the industry. We see in Chapter 8 that update protocols also have some other problems for scalable cache-coherent architectures, making it less attractive for microprocessors to support these protocols.

Having discussed how to keep data coherent, let us now consider how synchronization is managed in bus-based multiprocessors.

5.5 SYNCHRONIZATION

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations: mutual exclusion, point-to-point events, and global events. There has been considerable debate over the years about how much hardware support and exactly what hardware primitives should be provided to support these synchronization operations. The conclusions have changed from time to time with changes in technology and design style. Hardware support has the advantage of speed, but moving functionality to software has the advantages of cost, flexibility, and adaptability to different situations. The classic works of Dijkstra (1965) and Knuth (1966) show that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization methods rely on hardware support for some sort of *atomic read-modify-write* operation, in which the value of a memory location is ensured to be read, modified, and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built in software using these primitives.

The history of instruction sets offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare&swap* instruction, to support synchronization in concurrent programming on uniprocessor or multiprocessor systems. The compare&swap compares the value in a memory location with the value in a specified register and, if they are equal, swaps the value in the memory location with the value in a second specified register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic; since the source and destination operands are memory locations, much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high-level language architecture have proposed that the user-level synchronization operations, such as locks and barriers, should be supported directly at the machine level, not just atomic read-modify-write primitives; that is, the synchronization "algorithm" itself should be implemented in hardware. This issue became very active during the reduced instruction set debates since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The Sparc approach was to provide atomic operations involving a register or registers and a memory location using a simple swap (atomically swapping the contents of the specified register and memory location) and a compare&swap. MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described later in this section, which allows a variety of higher-level read-modify-write operations to be constructed without requiring the design to implement them all. In essence, the pair of instructions can be used instead of a sin-

gle instruction to implement atomic exchange or more complex atomic operations. This approach was later incorporated into the PowerPC and DEC Alpha architectures and is now quite popular. As we will see, synchronization brings to light a rich family of trade-offs across the layers of communication architecture. Not only can a spectrum of high-level operations and low-level primitives be supported by hardware, but the synchronization requirements of applications vary substantially as well.

The focus of this section is on how synchronization operations can be implemented on a bus-based cache-coherent multiprocessor through a combination of software algorithms and hardware primitives. In particular, it describes the implementation of mutual exclusion through lock-unlock pairs, point-to-point event synchronization through flags, and global event synchronization through barriers. Let us begin by considering the components of a synchronization event. This will make it clear why supporting the high-level mutual exclusion and event operations directly in hardware is difficult and is likely to make the implementation too rigid. Then, given that the hardware supports only the basic atomic operations, we can examine the role of the user software and system software in synchronization operations and then consider the hardware and software design trade-offs in greater detail.

5.5.1 Components of a Synchronization Event

There are three major components of a synchronization event:

1. *Acquire method:* a method by which a process tries to acquire the right to the synchronization (to enter the critical section or proceed past the event synchronization).
2. *Waiting algorithm:* a method by which a process waits for a synchronization to become available; for example, if a process tries to acquire a lock but the lock is not free, or to proceed past an event but the event has not yet occurred.
3. *Release method:* a method for a process to enable other processes to proceed past a synchronization event; for example, an implementation of the Unlock operation, a method for the last process arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

The choice of waiting algorithm is quite independent of the type of synchronization. There are two main choices: busy-waiting and blocking. *Busy-waiting* means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the waiting process to proceed. Under *blocking*, the process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait. It will be awakened and made ready to run again when the release it was waiting for occurs. The trade-offs between busy-waiting and blocking are clear. Blocking has higher overhead since suspending and resuming a process involves the operating system (and suspending and resuming a thread involves the run-time system of a threads package), but it makes the processor available to other

threads or processes that have useful work to do. Busy-waiting avoids the cost of suspension but consumes the processor and cache bandwidth while waiting. Blocking is strictly more powerful than busy-waiting because, if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.[7] Busy-waiting is likely to be better when the waiting period is short, whereas blocking is likely to be a better choice if the waiting period is long and if there are other processes to run. Hybrid waiting methods can be used in which the process busy-waits for a while in case the waiting period is short, and if the waiting period exceeds a certain threshold, the process blocks, allowing other processes to run (a *two-phase* waiting algorithm).

The difficulty in implementing high-level synchronization operations in hardware is not the acquire or the release component but the waiting algorithm. Thus, it makes sense to provide hardware support for the critical aspects of the acquire and release methods and allow the three components to be glued together in software. However, subtle but very important hardware/software interactions remain in how the spinning operation in the busy-wait component is realized.

5.5.2 Role of the User and System

Who should be responsible for implementing the internals of high-level synchronization operations such as locks and barriers? Typically, a programmer wants to use locks, events, or even higher-level operations without having to worry about their internal implementation. The implementation is left to the system, which must decide how much hardware support to provide and how much of the functionality to implement in software. Software synchronization algorithms using simple atomic exchange primitives have been developed that approach the speed of full hardware implementations, and the flexibility and hardware simplification they afford are very attractive. As with other aspects of system design, the utility of faster operations with more hardware support depends on the frequency of the use of those operations in the applications. So, once again, the best answer will be determined by a better understanding of application behavior.

Software implementations of synchronization constructs are usually included in system libraries. Good synchronization library design can be quite challenging. One potential complication is that the same type of synchronization (lock, barrier), and even the same synchronization variable, may be used at different times under very different run-time conditions. For example, a lock may be accessed with low contention (a small number of processors, maybe only one, trying to acquire the lock at a time) or with high contention (many processors trying to acquire the lock at the same time). The different scenarios impose different performance requirements.

7. This problem of denying resources to the critical process or thread is one that is actually made simpler with more processors. When the processes are time-shared on a single processor, strict busy-waiting without preemption is sure to be a problem. If each process or thread has its own processor, it is guaranteed not to be a problem. Multiprogramming environments on a limited set of processors may fall somewhere in between.

Under high contention, most processes will spend time waiting, and the key requirement of a lock algorithm is that it provide high lock-unlock transfer bandwidth; under low contention, the key goal is to provide low latency for lock acquisition. Different algorithms may satisfy different requirements better, so we must either find a good compromise algorithm or provide different algorithms for each type of synchronization from which a user can choose. If we are lucky, a flexible library can at run time choose the best implementation for the situation at hand. Different synchronization algorithms may also rely on different basic hardware primitives, so some may be better suited to a particular machine than others. Under multiprogramming, process scheduling and other resource interactions can change the synchronization behavior of the processes in a parallel program. A more sophisticated algorithm that addresses multiprogramming effects may provide better performance in practice than a simple algorithm that has lower latency and higher bandwidth in the dedicated case. All of these factors make synchronization a critical point of hardware/software interaction.

5.5.3 Mutual Exclusion

Mutual exclusion (lock-unlock) operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is little contention for the lock but inefficient under high contention, whereas sophisticated algorithms that deal well with contention have a higher cost in the low-contention case. After a brief discussion of hardware locks, this section describes the simplest software algorithms for memory-based locks using atomic exchange instructions. Following this is a discussion of how these simple algorithms can be implemented by using the special load-locked and store-conditional instruction pairs to synthesize atomic exchange, in place of atomic exchange instructions themselves, and what the tradeoffs are. Next, we will look at more sophisticated algorithms that can be built using either method of implementing atomic operations.

Hardware Locks

Lock operations can be supported entirely in hardware, although this is not popular on modern bus-based machines. One option that was used on some older machines was to have a set of lock lines on the bus, each used for one lock at a time. The processor holding the lock asserts the line, and processors waiting for the lock wait for it to be released. A priority circuit determines which processor gets the lock next when there are multiple requestors. However, this approach was quite inflexible since only a limited number of locks can be in use at a time and the waiting algorithm is fixed (typically a form of busy-wait with abort after time-out). Usually, these hardware locks were used only by the operating system for specific purposes, one of which was to implement a larger set of software locks in memory. The CRAY Xmp provided an interesting variant of this approach. A set of registers was shared among

the processors, including a fixed collection of lock registers. Although the architecture made it possible to assign lock registers to user processes, with only a small set of such registers it was awkward to do so in a general-purpose setting, and in practice the lock registers too were used primarily to implement higher-level locks in memory.

Simple Software Lock Algorithms

Consider a lock operation used to provide atomicity for a critical section of code. For the acquire method, a process trying to obtain a lock must check that the lock is free and, if it is, then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A simple way of thinking about the lock acquire operation is that a process trying to obtain the lock should check if the variable is 0 and if so set it to 1, thus marking the lock busy; if the variable is 1 (lock is busy), then it should wait for the variable to turn to 0 using the waiting algorithm. An unlock operation should simply set the variable to 0 (the release method). The following are assembly-level instructions for this attempt at a lock and unlock. (In our pseudo-assembly notation, the first operand always specifies the destination if there is one.)

```
lock:  ld   register, location  /*copy location to register*/
       cmp  register, #0        /*compare with 0*/
       bnz  lock                /*if not 0, try again*/
       st   location, #1        /*store 1 into location to mark it locked*/
       ret                      /*return control to caller of lock*/
```

and

```
unlock: st location, #0         /*write 0 to location*/
        ret                     /*return control to caller*/
```

The problem with this lock, which is supposed to provide atomicity for the critical section that follows it, is that it needs (but lacks) atomicity in its own implementation. To illustrate this, suppose that the lock variable was initially set to 0 and two processes P_0 and P_1 execute the above assembly code implementations of the lock operation. Process P_0 reads the value of the lock variable as 0 and thinks it is free, so it proceeds past the branch instruction. Its next step is to set the variable to 1, marking the lock as busy, but before it can do this, process P_1 reads the variable as 0, thinks the lock is free, and passes the branch instruction too. We now have two processes simultaneously proceeding past the lock and entering the same critical section, which is exactly what the lock was meant to avoid. Putting the store instruction just after the load instruction would not help either. The two-instruction sequence—reading (testing) the lock variable to check its state and writing (setting) it to busy if it is free—is not atomic, and there is nothing to prevent these operations in different processes from being interleaved in time. What we need is a way to atomically test the value of a variable and set it to another value if the test succeeds (i.e., to atomically read and then conditionally modify a memory location) and then

to return whether the atomic sequence was executed successfully or not. One way to provide this atomicity for user processes is to place the lock routine in the operating system and access it through a system call, but this is expensive and leaves the question of how the locks are supported by the operating system itself. Another option is to utilize a hardware lock around the instruction sequence for the lock routine, but this requires hardware locks and tends to be slow on modern processors.

An efficient, general-purpose solution to the lock problem is to support an atomic read-modify-write instruction in the processor's instruction set. A typical approach is to have an atomic exchange instruction: a value at a memory location specified by the instruction is read into a register, and another value is stored into the location, all in an atomic operation with no other accesses to that location allowed to intervene. Many variants of this operation exist with varying degrees of flexibility in the nature of the value that can be stored. A simple example that works for mutual exclusion is an atomic *test&set* instruction. In this case, the value in the memory location is read into a specified register, and the constant 1 is stored into the location atomically. The success of the test&set is determined by examining the value in the register. If it is 0, the test&set was successful. If it is 1, it was not successful; the value 1 written to memory by the test&set instruction is the same as was already there, so no harm is done. (1 and 0 are the values typically used, though any other constants might be used in their place.) Given such an instruction, with the mnemonic t&s, we can write a lock and unlock in pseudo-assembly language as follows:

```
lock:   t&s register, location
                              /*copy location to reg, and set location to 1*/
        bnz register, lock    /*compare old value returned with 0*/
                               /*if not 0, i.e., lock already busy, so try again*/
        ret                   /*return control to caller of lock*/
```

and

```
unlock:st location, #0        /*write 0 to location*/
        ret                   /*return control to caller*/
```

The lock implementation keeps trying to acquire the lock using test&set instructions until the test&set leaves zero in the register, indicating that the lock was free when tested (in which case the test&set has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the location associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic test&set instruction.

More sophisticated variants of such atomic instructions exist and, as we will see, are used by different software synchronization algorithms. One example is a *swap* instruction. Like a test&set, this reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location, it writes whatever value was in the register to begin with. That is, it atomically exchanges or swaps the values in the memory location and the register. Clearly,

we can implement a lock as before by replacing the test&set with a swap instruction as long as we use the values 0 and 1 and ensure that the value in the register is 1 before the swap instruction is executed; the lock has succeeded if the value left in the register by the swap instruction is 0.

Another example is the family of *fetch&op* instructions. A fetch&op instruction also specifies a location and a register. It atomically reads the current value of the location into the register and writes the value (which has been obtained by applying the operation specified by the fetch&op instruction to the current value of the location) into the location. The simplest forms of fetch&op to implement are the *fetch&increment* and *fetch&decrement* instructions, which change the current value by 1. A *fetch&add* would take another operand, which is a register or value, to add into the previous value of the location. A more complex primitive is the *compare&swap* operation. It takes two register operands and a memory location (i.e., it is a three-operand instruction, not commonly supported by RISC architectures); it compares the value in the location with the contents of the first register operand, and, if the two are equal, it swaps the contents of the memory location with the contents of the second register.

Performance of the Simple Lock

Figure 5.29 shows the performance of a simple test&set lock on the SGI Challenge.[8] Performance is measured for the following microbenchmark executed repeatedly in a loop:

```
lock(L);
critical-section(c);
unlock(L);
```

where c is a delay parameter that determines the size of the critical section (it is only a delay in this case, with no real work done). The benchmark is configured so that the same total number of lock calls are executed as the number of processors increases, reflecting a situation where a fixed number of tasks must be dequeued from a centralized task queue, independent of the number of processors. Performance is measured as the time per lock transfer, that is, the cumulative time taken by all processes executing the benchmark divided by the number of times the lock is obtained. The cumulative time spent in the critical section itself (i.e., c times the number of successful locks executed) is subtracted from the cumulative execution time so that only the time for the lock transfers themselves (or any contention caused by the lock operations) is obtained. All measurements are in microseconds.

8. In fact, the processor on the SGI Challenge, which is the machine for which synchronization performance is presented in this chapter, does not provide a test&set instruction. Rather, it uses alternative primitives that will be described later in this section. For these experiments, a mechanism whose behavior closely resembles that of test&set is synthesized from the available primitives. Results for real test&set-based locks on older machines like the Sequent Symmetry can be found in the literature (Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991).

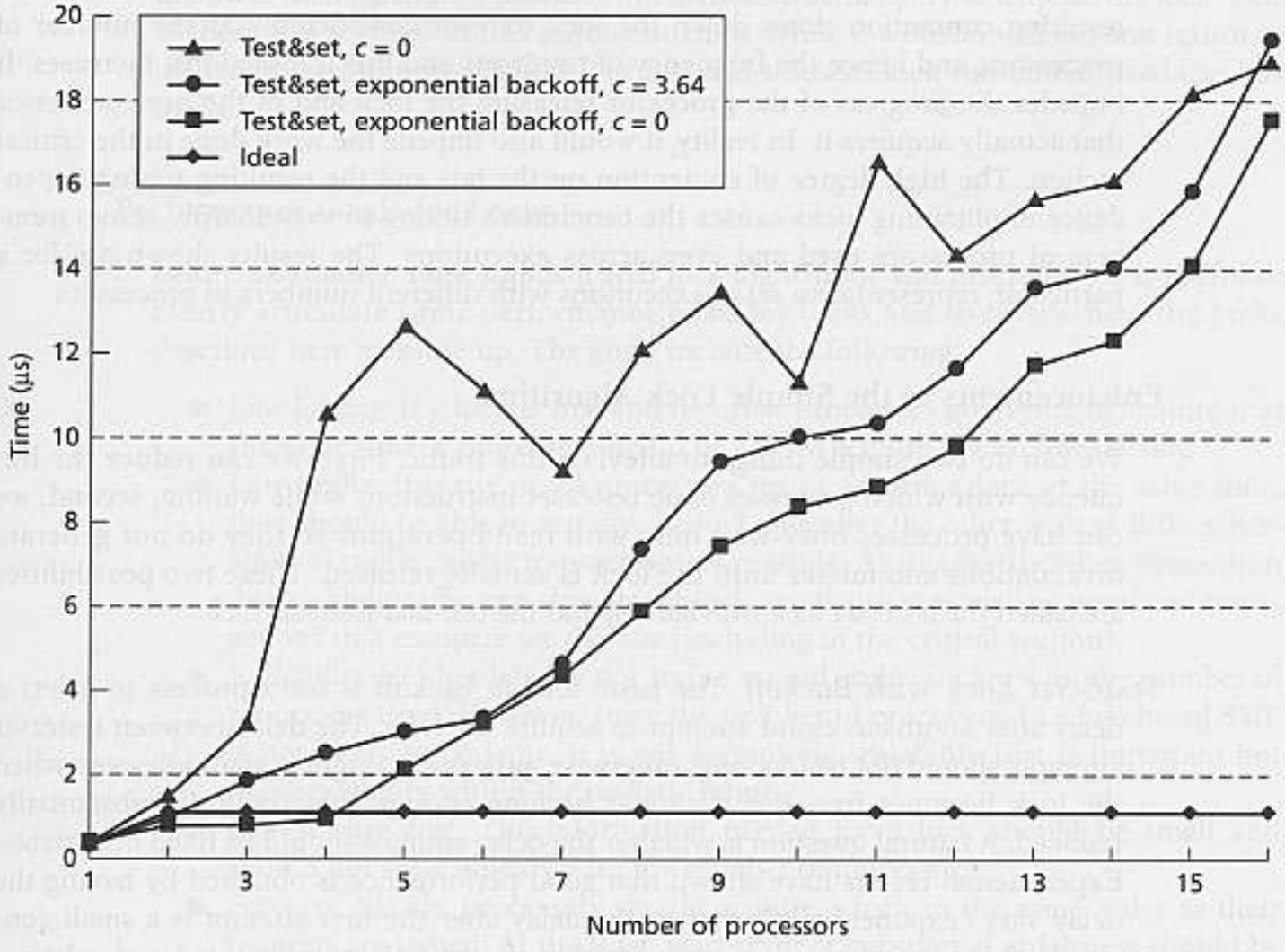

FIGURE 5.29 Performance of the synthesized test&set locks with an increasing number of competing processors on the SGI Challenge. The *y*-axis is the time per lock-unlock pair, excluding the critical section of size c microseconds. The irregular nature of the top curve is due to the timing dependence of the contention effects caused.

The upper curve in the figure shows the time per lock transfer with an increasing number of processors when using the test&set lock with a very small critical section (ignore the curves with "backoff" in their labels for now). Ideally, we would like the time per lock acquisition to be independent of the number of processors competing for the lock, with only one uncontended bus transaction per lock transfer, as shown in the curve labeled "ideal." However, the figure shows that performance clearly degrades with an increasing number of processors.

The problem with the test&set lock is the traffic generated during the waiting method: every attempt to check whether the lock is free to be acquired, whether successful or not, generates a write operation to the cache block that holds the lock variable (since it uses a test&set operation and writes the value to 1); since this block is currently in the cache of some other processor (which wrote it last when doing its test&set), a bus transaction is generated by each write to invalidate the previous owner of the block. Thus, all processors put transactions on the bus repeat-

edly and consume precious bus bandwidth even during the waiting algorithm. The resulting contention slows down the lock transfer considerably as the number of processors, and hence the frequency of test&sets and bus transactions, increases. It impedes the progress of the processor releasing the lock and of the next processor that actually acquires it. In reality, it would also impede the work done in the critical section. The high degree of contention on the bus and the resulting timing dependence of obtaining locks causes the benchmark timing to vary sharply across numbers of processors used and even across executions. The results shown are for a particular, representative set of executions with different numbers of processors.

Enhancements to the Simple Lock Algorithm

We can do two simple things to alleviate this traffic. First, we can reduce the frequency with which processes issue test&set instructions while waiting; second, we can have processes busy-wait only with read operations so they do not generate invalidations and misses until the lock is actually released. These two possibilities are called the *test&set lock with backoff* and the *test-and-test&set lock*.

Test&Set Lock with Backoff The basic idea in backoff is for a process to insert a delay after an unsuccessful attempt to acquire the lock. The delay between test&set attempts should not be too long; otherwise, processors might remain idle even when the lock becomes free. But it should be long enough that traffic is substantially reduced. A natural question is whether the delay amount should be fixed or variable. Experimental results have shown that good performance is obtained by having the delay vary "exponentially"; that is, the delay after the first attempt is a small constant k that increases geometrically, so that after the ith attempt, it is $k \times c^i$, where c is another constant. Such a lock is called a test&set lock with exponential backoff. Figure 5.29 also shows the performance for the test&set lock with backoff for two different sizes of the critical section, using the starting value k for backoff that appears to perform best. Performance improves but still does not scale very well since there is still substantial traffic interfering with the release and acquire. Performance results using backoff with a real test&set instruction on older machines can be found in the literature (Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991). See also Exercise 5.14, which discusses why the performance with a nonzero critical section is worse than that with a null critical section when backoff is used.

Test-and-Test&Set Lock A more subtle change to the algorithm is to have it use instructions that do not generate as much bus traffic while busy-waiting. Processes busy-wait by repeatedly reading with a standard load, not a test&set, the value of the lock variable until it turns from 1 (locked) to 0 (unlocked). On a cache-coherent machine, the reads can be performed in-cache by all processors, without generating bus traffic, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes are invalidated, and the next read of the variable by each process will generate a read miss. The waiting processes will then find that the lock has been made available and only

then will each generate a test&set instruction to actually try to acquire the lock. One of them will succeed in this acquire attempt, while the others will fail and return to the read-based waiting method. The test-and-test&set lock substantially reduces bus traffic.

Performance Goals for Locks

Before examining more sophisticated lock algorithms and primitives, it is useful to clearly articulate some performance goals for locks and to review how the locks described here measure up. The goals include the following:

- *Low latency.* If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.
- *Low traffic.* If many or all processors try to acquire a lock at the same time, they should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible. As discussed earlier, contention due to high traffic can slow down lock acquisitions as well as unrelated transactions that compete for the bus (including in the critical section).
- *Scalability.* Neither latency nor traffic should scale quickly with the number of processors used. However, since the number of processors in a bus-based SMP is not likely to be large, it is not asymptotic scalability that is important but only scalability within the realistic range.
- *Low storage cost.* The information needed for a lock should be small and should not scale quickly with the number of processors.
- *Fairness.* Ideally, processors should acquire a lock in the same order as their requests are issued. At the least, starvation or substantial unfairness should be avoided. Since starvation is usually unlikely, the importance of fairness must be traded off with its impact on performance.

Consider the simple atomic exchange or test&set lock. It is very low latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, we have seen that it can generate a lot of bus traffic and contention if many processors compete for the lock. The performance of the lock scales poorly as the number of competing processors increases. The storage cost is low (a single variable suffices) and does not scale with the number of processors. The lock makes no attempt to be fair, and an unlucky processor can be starved out. The test&set lock with backoff has the same uncontended latency as the simple test&set lock, generates less traffic, is somewhat more scalable, takes no more storage, and is no more fair. The test-and-test&set lock has slightly higher uncontended latency than the simple test&set lock (it does a read in addition to a test&set even when there is no competition) but generates much less bus traffic and is more scalable. It too requires negligible storage and is not fair. (Exercise 5.12 asks you to count the number of bus transactions and the time required for the test-and-test&set type of lock in different scenarios.)

In the test-and-test&set lock, since a test&set operation (and hence a bus transaction) is only issued when a processor is notified that the lock is ready, and thereafter if it fails it busy-waits (spins) on a cached block, there is no need for backoff. However, the lock does have the problem that when the lock is released, all waiting processes rush out and perform their read misses and their test&set instructions at about the same time. The bus transactions for the read misses may be combined in a smart bus protocol; however, each of the test&set instructions itself generates invalidations and subsequent misses, resulting in $O(p^2)$ bus traffic for p processors to acquire the lock once each. A random delay before issuing the test&set could help to stagger at least the test&set instructions, but it would increase the latency to acquire the lock in the uncontended case. While test-and-test&set was a major step forward at its time, better hardware primitives and better algorithms have been designed to alleviate its traffic problem.

Improved Hardware Primitives: Load-Locked, Store-Conditional

In addition to spinning with reads rather than read-modify-writes, which test-and-test&set accomplishes, we would prefer that failed attempts to complete the read-modify-write do not generate invalidations. It would also be useful to have a single primitive that allows us to implement a range of atomic read-modify-write operations—such as test&set, fetch&op, compare&swap—rather than implementing each with a separate instruction. One way to achieve both goals, increasingly supported in modern microprocessors, is to use a pair of special instructions rather than a single read-write-modify instruction to implement atomic access to a variable (let's call it a synchronization variable). The first instruction, commonly called *load-locked* or *load-linked* (LL), loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register—that is, the modify part of a read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional*. It tries to write the register back to the memory location (the synchronization variable) if and only if no other processor has written to that location (or cache block) since this processor completed its LL. Thus, if the store-conditional succeeds, it means that the load-locked, store-conditional (LL-SC) pair has read, perhaps modified in between, and written back the variable atomically. If the store-conditional detects that an intervening write has occurred to the variable or cache block, it fails and does not even try to write the value back (or generate any invalidations). This means that the atomic operation on the variable has failed and must be retried starting from the LL. Success or failure of the store-conditional is indicated by the condition codes or a return value. How the LL and store-conditional are actually implemented will be discussed later; for now, we are concerned with their semantics and performance.

Using LL-SC to implement atomic operations, the simple lock and unlock algorithms can be written as follows, where `reg1` is the register into which the current value of the memory location is loaded and `reg2` holds the value to be stored in the memory location by this atomic exchange (`reg2` could simply be the value 1 for a lock attempt, as in a test&set).

```
lock:   ll   reg1, location     /*load-locked the location to reg1*/
        bnz  reg1, lock         /*if location was locked (nonzero),
                                  try again*/
        sc   location, reg2     /*store reg2 conditionally into location*/
        beqz lock               /*if store-conditional failed, start again*/
        ret                     /*return control to caller of lock*/
```

and

```
unlock: st location, #0         /*write 0 to location*/
        ret                     /*return control to caller*/
```

Many processors may perform the LL at the same time, but only the first one that manages to put its store-conditional on the bus will actually succeed in its store-conditional. This processor will have succeeded in acquiring the lock, whereas the others will have failed and will have to retry the LL-SC. Note that the store-conditional may fail either because it detects the occurrence of an intervening write before even attempting to access the bus or because it attempts to get the bus but some other processor's store-conditional gets there first. Of course, if the location is 1 (nonzero) when a process does its LL, it will load 1 into `reg1` and will retry the lock starting from the LL without even attempting the store-conditional.

It is worth noting that the LL itself is not a lock and the store-conditional itself is not an unlock. For one thing, the completion of the LL itself does not imply obtaining exclusive access; in fact, LL and store-conditional are used together to implement a lock operation. For another, even a successful LL-SC pair does not guarantee that the instructions between them (if any) are executed atomically with respect to those instructions on other processors, so in fact these instructions do not constitute a critical section. All that a successful LL-SC guarantees is that no conflicting writes to the synchronization variable itself intervene between the LL and store-conditional. In fact, since the instructions between the LL and store-conditional are executed unconditionally but should not be visible if the store-conditional fails, it is important that they do not modify any other important state. Typically, these instructions manipulate only the register into which the synchronization variable is loaded—for example, to perform the op part of a fetch&op—and do not modify any other program variables (modification of this register is okay since the register will be reloaded anyway by the LL in the next attempt). Microprocessor vendors that support LL-SC explicitly encourage software writers to follow this guideline and, in fact, often specify what instructions are possible to insert with a guarantee of correctness given their implementations of LL-SC. The number of instructions between the LL and store-conditional should also be kept small to reduce the probability of store-conditional failure due to an intervening write. Although the LL and store-conditional do not constitute a lock-unlock pair, they can be used directly to implement certain atomic operations on shared data structures. For example, if the desired function is a small operation on a globally shared variable (like a counter or global sum), it makes much more sense to implement it as the natural sequence (LL, register op, store-conditional, test) than to build a lock and unlock around the variable update.

Like the test-and-test&set, the spin-lock built with LL-SC does not generate bus traffic during the waiting algorithm if the LL indicates that the lock is currently held. Better than the test-and-test&set, it also does not generate invalidations on a failed attempt to obtain the lock (i.e., a failed store-conditional). However, when the lock is released, the processors spinning in a tight loop of load-locked operations will indeed miss on the location and rush out to the bus with read transactions. After this, only a single invalidation will be generated for a given lock acquisition by the processor whose store-conditional succeeds, but this will again invalidate all caches. Traffic is reduced greatly from even the test-and-test&set case and there are no read-modify-write bus transactions, but traffic still increases linearly with the number of processors (i.e., $O(p)$ bus transactions per look acquisition). Since spinning on a locked location is already done through reads (load-locked operations), no analog of a test-and-test&set exists to further improve its performance. However, backoff can be used between the LL and store-conditional to reduce bursty traffic.

The simple LL-SC lock is also low in latency and storage, but it is not a fair lock and does not reduce traffic to a minimum. More advanced lock algorithms can be used that provide both fairness and reduced traffic. They can be built using either atomic read-modify-write instructions or atomic operations of equivalent semantics synthesized with LL-SC, though of course the traffic advantages are different in the two cases. Let us consider two of these algorithms that are appropriate for bus-based machines.

Advanced Lock Algorithms

Especially when using an atomic exchange instruction like test&set, instead of LL-SC, to implement locks, it is desirable to have only one process actually attempt to obtain the lock when it is released (rather than have them all rush out to do a test&set and issue invalidations as in all the preceding algorithms). It is even more desirable to have only one process incur a read miss (even with LL-SC) when a lock is released. The *ticket lock* accomplishes the first purpose; the *array-based lock* accomplishes both goals but at a little cost in space. Unlike all the previous locks, both these locks are fair and grant the lock to processors in FIFO order.

Ticket Lock The ticket lock operates just like the ticket system in the sandwich line at a delicatessen or like the teller line at a bank. Every process wanting to acquire the lock takes a ticket number and then busy-waits on a global `now-serving` number—like the number on the LED display that we watch intently in the sandwich line—until the `now-serving` number equals the ticket number it obtained. To release the lock, a process simply increments the `now-serving` number so that the next waiting process can acquire the lock. The atomic primitive needed is a fetch&increment, which a process uses when it first reaches the lock operation to obtain its ticket number from a shared counter. No atomic operation (e.g., test&set) is needed to actually obtain the lock upon a release since only the unique process that has its ticket number equal to `now-serving` attempts to enter the critical section when it sees the release. Thus, the acquire method is the fetch&increment, the

waiting algorithm is busy-waiting for `now-serving` to equal the ticket number, and the release method is to increment `now-serving`. This lock has uncontended latency about equal to the test-and-test&set lock but generates much less traffic. Although every process does a fetch&increment when it first arrives at the lock (presumably not every process at the same time), the test&set attempts upon a release of the lock are eliminated, which tend to be simultaneous and a lot more heavily contended. The ticket lock also requires constant and small storage and is fair since processes obtain the lock in the order of their fetch&increment operations.

The fetch&increment needed by the ticket lock can be implemented with LL-SC. However, since the simple LL-SC lock already avoids multiple processors issuing invalidations in trying to acquire a lock after its release, there is not a large difference in traffic between the ticket lock and the simple LL-SC lock. (The simple LL-SC lock is somewhat worse since in that case another invalidation and set of read misses occur when a processor succeeds in its store-conditional.) The key difference between these two locks is fairness.

Like the simple LL-SC lock, the ticket lock still has a read traffic problem at a release. The reason is that all processes spin on the same variable (`now-serving`). When that variable is written at a release, all processors' cached copies are invalidated, and they all incur a read miss. The read misses may be combined on some buses but can cause unnecessary traffic if the combining is unavailable or unsuccessful. One way to reduce this bursty read-miss traffic is to introduce a form of backoff. We do not want to use exponential backoff because we do not want all processors to be backing off when the lock is released so that none tries to acquire it for a while. A promising technique is to have each processor back off from trying to read the `now-serving` counter by a duration proportional to when it expects its turn to actually come—that is, by a duration proportional to the difference in its ticket number and the `now-serving` value it last read. Alternatively, the array-based lock completely eliminates this extra read traffic upon a release by having every process spin on a distinct location.

Array-Based Lock The idea here is to use a fetch&increment to obtain not a value but a unique location on which to busy-wait. If there are p processes that might possibly compete for a lock, then the lock data structure contains an array of p locations that processes can spin on, ideally each on a separate memory block to avoid false sharing. The acquire method then uses a fetch&increment operation to obtain the next available location in this array (with wraparound), the waiting method spins on this location, and the release method writes a value denoting "unlocked" to the next location in the array (after the one that the releasing processor was itself spinning on). Only the processor that was spinning on that next location has its cache block invalidated at the release; its consequent read miss tells it that it has obtained the lock. As in the ticket lock, no test&set is needed after the miss since only one process is notified when the lock is released. This lock is clearly also FIFO and hence fair. Its uncontended latency is likely to be similar to that of the test-and-test&set lock (a fetch&increment followed by a read of the assigned array location), and it is potentially more scalable than the ticket lock since only one processor incurs the

read miss. For the same reason, unlike the ticket lock, it does not need any form of backoff to reduce traffic. Its only drawback for a bus-based machine is that it uses $O(p)$ space rather than $O(1)$, but with both p and the proportionality constant being small, this is usually not a very significant drawback. It has a potential drawback for machines with distributed memory, but we shall discuss this drawback and lock algorithms that overcome it in Chapter 7.

Performance

Let us briefly examine the performance of the different locks on the SGI Challenge, as shown in Figure 5.30. All locks are implemented using LL-SC since the Challenge provides only these and not atomic instructions. Results are shown for a somewhat more parameterized version of the earlier microbenchmark code, in which a process is allowed to insert a delay not only for the critical section but also between its release of the lock and its next attempt to acquire it (as will happen in a real program). That is, the code is a loop over the following body:

```
lock(L);
critical_section(c);
unlock(L);
delay(d);
```

Let us consider three cases: (1) $c = 0$, $d = 0$; (2) $c = 3.64$ µs, $d = 0$; and (3) $c = 3.64$ µs, $d = 1.29$ µs—called the *null* critical section case, the *non-null* critical section case, and the non-null critical section with *delay* case, respectively. The delays c and d are inserted in the code as round numbers of processor cycles, which translates to these microsecond numbers. Recall that in all cases, the delays c and d (multiplied by the number of lock acquisitions by each processor) are subtracted out of the total time, which is supposed to measure only the total time taken for a certain number of lock acquisitions and releases (see also Exercise 5.15).

Consider the null critical section case. The first observation, comparing Figure 5.30 with Figure 5.29, is that all the other locks are indeed better than the test&set locks, as expected.[9] The second observation is that the simple LL-SC locks actually seem to perform better than the more sophisticated ticket lock and array-based lock. For these locks, which don't encounter as much contention as the test&set lock, performance is largely determined by the number of bus transactions between a release and a successful acquire. The reason that the LL-SC locks perform so well, particularly at lower processor counts, is that they are not fair, and the unfairness is exploited by architectural interactions! In particular, when a processor that releases a lock with a write follows it immediately with the read (LL) for its next acquire, its read and the subsequent store-conditional are likely to succeed in its cache before

9. The test&set is simulated using LL-SC as follows: every time a store-conditional fails, a write is performed to another variable in the same cache block, causing invalidations as a test&set would. This method of simulating test&set with LL-SC may lead to somewhat worse performance than a true test&set primitive, but it conveys the trend.

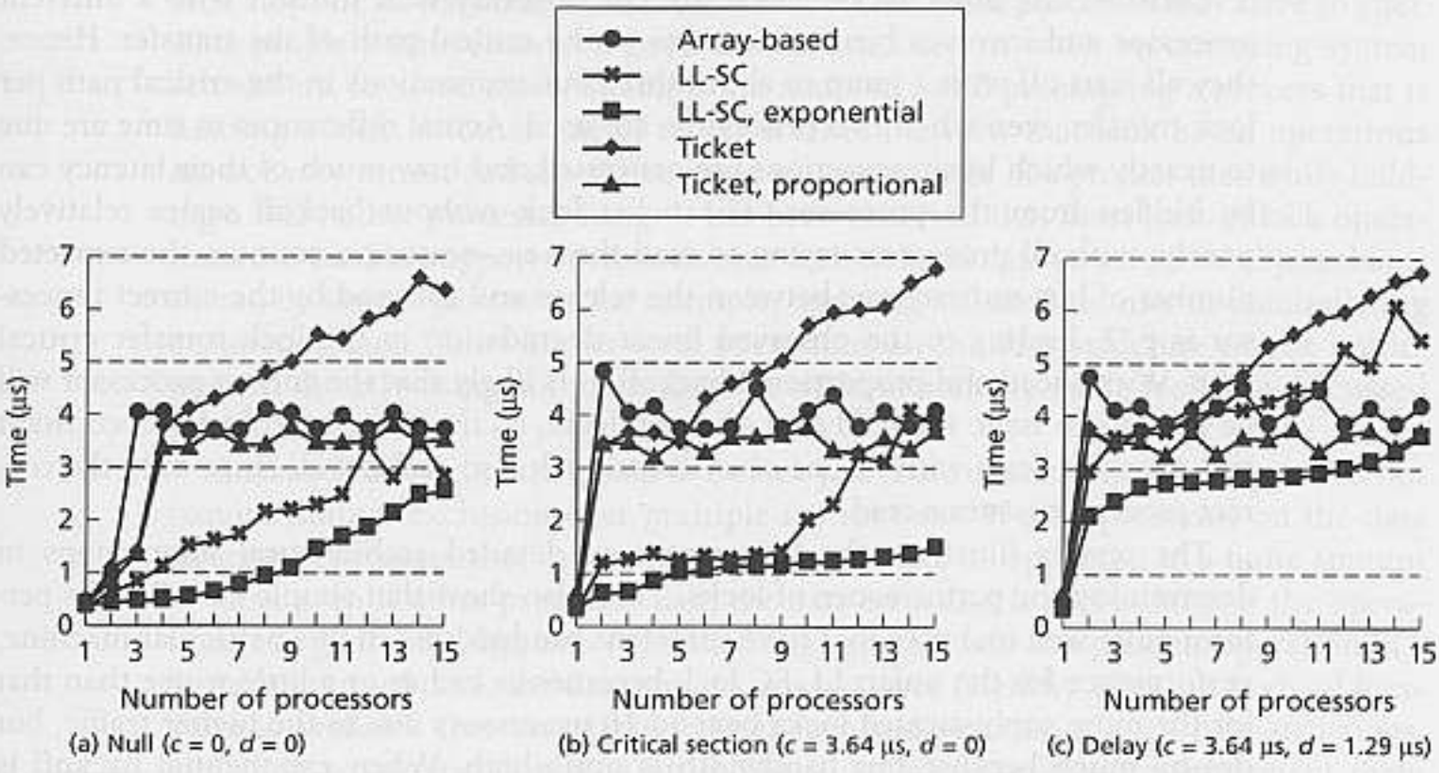

FIGURE 5.30 Performance of locks on the SGI Challenge for three different scenarios

another processor can read the block across the bus. (The bias on the SGI Challenge is actually more severe, since the releasing processor can satisfy its next read from its write buffer even before the read exclusive corresponding to the releasing write gets out on the bus.) Lock transfer is very quick, and performance is good, but the same processor keeps acquiring the lock repeatedly. As the number of processors and the competition for the bus increase, the likelihood of the last releaser's store-conditional successfully obtaining the bus decreases, and hence the likelihood of self-transfers decreases. In addition, bus traffic increases due to invalidations and read misses, so the time per lock transfer increases. Exponential backoff helps reduce the burstiness of traffic and hence slows the rate of scaling, and a nonzero critical section ($c = 3.64$, $d = 0$) helps this along further.

With delays both inside and outside the critical section ($c = 3.64$, $d = 1.29$), we see the LL-SC lock not doing quite as well, even at low processor counts. This is because a processor waits after its release before trying to acquire the lock again, making it much more likely that some other waiting processor will acquire the lock before it. Self-transfers are unlikely, so lock transfers are slower even with two processors. It is interesting that performance is particularly worse for the backoff case at small processor counts when the delay d between unlock and lock is nonzero. This is because it is quite likely that while the processor that just released the lock is waiting for d to expire before doing its next acquire, all the other processors are in a backoff period and not even trying to acquire the lock. In the $d = 0$ case, the releasing processor reacquires the lock right away, especially with a small number of processors. Backoff must be used carefully for it to be successful.

Consider the other locks. These are fair, so every lock transfer is to a different processor and involves bus transactions in the critical path of the transfer. Hence, they all start off with a jump to about three bus transactions in the critical path per lock transfer even when two processors are used. Actual differences in time are due to exactly which bus transactions are generated and how much of their latency can be hidden from the processor. The ticket lock without backoff scales relatively poorly: with all processors trying to read the `now-serving` counter, the expected number of bus transactions between the release and the read by the correct processor is $p/2$, leading to the observed linear degradation in the lock transfer critical path. With successful proportional backoff, it is likely that the correct processor will be the one to issue the read first after a release, so the time per transfer is constant and does not scale with p. The array-based lock also scales well since only the correct processor issues a read.

The results illustrate the importance of detailed architectural interactions in determining the performance of locks. They also show that simple LL-SC locks perform quite well on buses that have sufficient bandwidth. On this particular machine, performance for the unfair LL-SC lock becomes as bad as or a little worse than that for the more sophisticated locks beyond 16 processors due to the higher traffic, but not by much because bus bandwidth is quite high. When exponential backoff is used to reduce traffic, the simple LL-SC lock delivers the best average lock transfer time in all cases. However, these results also illustrate the difficulty and the importance of sound experimental methodology in evaluating synchronization algorithms. Null critical sections display some interesting effects, but meaningful comparisons depend on what the synchronization patterns look like in practice—in real applications. For example, the effect of critical section and delay size on the frequency of self-transfers has a substantial impact on the comparison of unfair locks with fair locks. The nonrepresentativeness of the null case in this regard is therefore an important methodological consideration. An experiment to use LL-SC while guaranteeing round-robin acquisition among processors (fairness) by using an additional variable showed performance very similar to that of the ticket lock, confirming that unfairness and self-transfers are indeed the reason for the better performance at low processor counts. Especially if fairness is desired, the ticket lock with proportional backoff and the array-based lock perform very well on bus-based machines.

Lock-Free, Nonblocking, and Wait-Free Synchronization

An additional set of performance concerns involving synchronization arises when we consider that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time or, even if we have the machine to ourselves, background daemons run periodically, processes take page faults, I/O interrupts occur, and the process scheduler makes scheduling decisions with limited information about the application requirements. These events can cause the rate at which processes make progress to vary considerably. One important question is how the parallel program as a whole slows down when one process is slowed. With traditional locks, the problem can be serious: if a process holding a

lock stops or slows while in its critical section, all other processes may have to wait. This problem has received a good deal of attention in work on operating system schedulers. In some cases, attempts are made to avoid preempting a process that is holding a lock. Another line of research takes the view that lock-based operations are not very robust and should be avoided; for example, if a process dies while holding a lock, other processes hang. It has been observed that most lock-unlock operations are used to support operations on a well-defined data structure or object that is shared by several processes, for example, updating a shared counter or manipulating a shared queue. These higher-level operations on the data structure can be implemented directly using atomic primitives without actually using locks, as discussed for LL-SC earlier.

A shared data structure is said to be *lock-free* if the operations defined on it do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in a finite amount of time, even if other processes halt, the data structure is *nonblocking*. If the operations can guarantee that every (nonfaulting) process will complete its operation in a finite amount of time, the data structure is *wait-free* (Herlihy 1993). A body of literature is available that investigates the theory and practice of such data structures, including requirements placed on the basic atomic primitives to implement them (Herlihy 1988), general-purpose techniques for translating sequential operations to nonblocking concurrent operations (Herlihy 1993), specific useful lock-free data structures (Valois 1995; Michael and Scott 1996), operating system implementations (Massalin and Pu 1991; Greenwald and Cheriton 1996), and proposals for architectural support (Herlihy and Moss 1993). The basic approach is to implement updates to a shared object by reading a portion of the object to make a copy, updating the copy, and then performing an operation to commit the change only if no conflicting updates have been made (reminiscent of LL-SC). As a simple example, consider a shared counter. The counter is read into a register, a value is added to the register copy, and the result is put in a second register. Next, a compare&swap updates the shared counter only if its value is still the same as the copy. For more sophisticated, linked-list data structures, a new element is created and then linked into the shared list if the insert is still valid. These techniques serve to limit the window in which the shared data structure is in an inconsistent state, so they improve robustness, although it can be difficult to make them efficient.

Theoretical research has identified the properties of different atomic exchange operations in terms of the time complexity of using them to implement synchronized access to variables. In particular, it has been found that simple operations like test&set and fetch&op are not powerful enough to guarantee that the time taken by a processor to access a synchronized variable is independent of the number of processors, whereas more sophisticated atomic operations like compare&swap and swapping the values of two memory locations are powerful enough to make this guarantee (Herlihy1988).

Having discussed the options for mutual exclusion on bus-based machines, let us move on to point-to-point, and then barrier, event synchronization.

5.5.4 Point-to-Point Event Synchronization

Point-to-point synchronization within a parallel program is often implemented by busy-waiting on ordinary variables, using them as flags. If we want to use blocking instead of busy-waiting, we can use semaphores, just as they are used in concurrent programming and operating systems (Tanenbaum and Woodhull 1997).

Software Algorithms

Flags are control variables, typically used to communicate the occurrence of a synchronization event rather than to transfer values. If two processes have a producer-consumer relationship on the shared variable *a*, then a flag can be used to manage the synchronization as follows:

P_1		P_2
`a = f(x);`	*/*set a*/*	`while (flag is 0) do nothing;`
`flag = 1;`		`b = g(a);` */*use a*/*

If we know that the variable *a* is initialized to a certain value (say, 0), which will be changed to a new value we are interested in by this production event, then we can use *a* itself as the synchronization flag, as follows:

P_1		P_2
`a = f(x);`	*/*set a*/*	`while (a is 0) do nothing;`
		`b = g(a);` */*use a*/*

This eliminates the need for a separate flag variable and saves the write to and read of that variable at perhaps some cost in readability and maintainability.

Hardware Support: Full-Empty Bits

This idea of special flag values has been extended in some research machines (although mostly in machines with physically distributed memory) to provide hardware support for fine-grained producer-consumer synchronization. A bit, called a *full-empty bit*, is associated with every word in memory. This bit is set when the word is "full" with newly produced data (i.e., on a write) and unset when the word is "emptied" by a processor consuming that data (i.e., on a read). Word-level producer-consumer synchronization is then accomplished as follows. When the producer process wants to write the location, it does so only if the full-empty bit is set to empty and then leaves the bit set to full. The consumer reads the location only if the bit is full and then sets it to empty. Hardware preserves the atomicity of the read or write

with the manipulation of the full-empty bit. Given full-empty bits, our preceding example can be written without the spin loop as

P_1	P_2
`a = f(x);` /*set a*/	`b = g(a);` /*use a*/

Full-empty bits raise concerns about flexibility. For example, they do not lend themselves easily to single-producer-multiple-consumer synchronization or to the case where a producer updates a value multiple times before a consumer consumes it. Also, should all reads and writes use full-empty bits or only those that are compiled down to special instructions? The latter method requires support in the language and compiler, but the former is too restrictive in imposing synchronization on all accesses to a location (for example, it does not allow asynchronous relaxation in iterative equation solvers; see Chapter 2). For these reasons, and the hardware cost, full-empty bits have not found favor in most commercial machines.

Interrupts

Another important kind of event is the interrupt conveyed from an I/O device needing attention to a processor. In a uniprocessor machine, there is no question where the interrupt should go, but in an SMP any processor can potentially take the interrupt. In addition, there are times when one processor may need to issue an interrupt to another. In early SMP designs, special hardware was provided to monitor the priority of the process on each processor and to deliver the I/O interrupt to the processor running at lowest priority. Such measures proved to be of small value, and most modern machines use simple arbitration strategies. In addition, a memory-mapped interrupt control region usually exists, so at kernel level any processor can interrupt any other by writing the interrupt information at the associated address.

5.5.5 Global (Barrier) Event Synchronization

Finally, let us examine barrier synchronization on a bus-based machine. Software algorithms for barriers are typically implemented using locks, shared counters, and flags. Let us begin with a simple barrier among p processes, which is called a *centralized barrier* since it uses only a single lock, a single counter, and a single flag.

Centralized Software Barrier

A shared counter maintains the number of processes that have arrived at the barrier and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, a process checks to see if the counter equals p, that is, if it is the last process to have arrived. If not, it busy-waits

on the flag associated with the barrier; if so, it writes the flag to release the $p - 1$ waiting processes. A simple attempt at a barrier algorithm may therefore look like

```
struct bar_type {
  int counter;
  struct lock_type lock;
  int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
  LOCK(bar_name.lock);
  if (bar_name.counter == 0)
    bar_name.flag = 0;                    /*reset flag if first to reach*/
  mycount = bar_name.counter++;           /*mycount is a private variable*/
  UNLOCK(bar_name.lock);
  if (mycount == p) {                     /*last to arrive*/
    bar_name.counter = 0;                 /*reset counter for next barrier*/
    bar_name.flag = 1;                    /*release waiting processes*/
  }
  else
    while (bar_name.flag == 0) {}; /*busy-wait for release*/
}
```

Centralized Barrier with Sense Reversal

Can you see a problem with the preceding barrier? There is one. It occurs when the barrier operation is performed consecutively using the same barrier variable—for example, if each processor executes the following code:

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

The first process to enter the barrier the second time reinitializes the barrier counter, so that is not a problem. The problem is the flag. To exit the first barrier, processes spin on the flag until it is set to 1. Processes that see the flag change to 1 will exit the barrier, perform the subsequent computation, and enter the barrier again. However, suppose one processor P_x does not see the flag change from the first barrier before others have reentered the barrier for the second time; for example, it gets swapped out by the operating system because it has been spinning too long. When it is swapped back in, it will continue to wait for the flag to change to 1. In the meantime, other processes may have already entered the second instance of the barrier, and the first of these will have reset the flag to 0. Now the flag can only get set to 1

again when all p processes have registered at the new instance of the barrier, which will never happen since P_x will never leave the spin loop from the first barrier.

How can we solve this problem? What we need to do is prevent a process from entering a new instance of a barrier until all processes have exited the previous instance of the same barrier. One way is to use another counter to count the processes that leave the barrier and to not let a process reset the flag in a new barrier instance until this counter has turned to p for the previous instance. However, manipulating this counter incurs further latency and contention. On the other hand, with the current setup we cannot wait for all processes to reach the barrier before resetting the flag to 0, since that is when we actually set the flag to 1 for the release. A better solution is to avoid explicitly resetting the flag value altogether and rather have processes wait for the flag to obtain a different release value in consecutive instances of the barrier. For example, processes may wait for the flag to turn to 1 in one instance and to turn to 0 in the next instance. A private variable is used per process to keep track of which value to wait for in the current barrier instance. Since by the semantics of a barrier a process cannot get more than one barrier ahead of another, we only need two values (0 and 1) that we toggle between each time. Hence we call this method *sense reversal*. Now, in the previous example, the flag need not be reset when the first process reaches the barrier; rather, the process stuck in the old barrier instance still waits for the flag to reach the old release value while processes that enter the new instance wait for the other (toggled) release value. The value of the flag is only changed once when all processes have reached the (new) barrier instance, so it will not change before processes stuck in the old instance see it. Here is the code for a simple barrier with sense reversal:

```
BARRIER (bar_name, p)
{
    local_sense = !(local_sense);          /*toggle private sense variable*/
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;          /*mycount is a private variable*/
    if (bar_name.counter == p) {           /*last to arrive*/
        UNLOCK(bar_name.lock);
        bar_name.counter = 0;              /*reset counter for next barrier*/
        bar_name.flag = local_sense;       /*release waiting processes*/
    }
    else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {};  /*busy-wait for
                                                    release*/
    }
}
```

Note that the lock is not released immediately after the increment of the counter but only after the condition is evaluated; the reason for this is revealed in an exercise (see Exercise 5.18). We now have a correct barrier that can be reused any number of times consecutively. The remaining issue is performance, which we examine next.

(Note that the LOCK/UNLOCK protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation.)

Performance

The major performance goals for a barrier are similar to those for locks. They include the following:

- *Low latency (small critical path length).* The chain of dependent operations and bus transactions needed for p processors to pass the barrier should be small.
- *Low traffic.* Since barriers are global operations, it is quite likely that many processors will try to execute a barrier at the same time. The barrier algorithm should reduce the total number of bus transactions (whether in the critical path or not) and hence the possible contention.
- *Scalability.* Latency and traffic should increase slowly with the number of processors.
- *Low storage cost.* We would, of course, like to keep the storage cost low.
- *Fairness.* We should ensure that the same processor does not always become the last one to exit the barrier (or we may want to preserve FIFO ordering).

In the centralized barrier described previously, each processor accesses the lock once, hence the critical path length is at least proportional to p. Consider the bus traffic. To complete its operation, a centralized barrier involving p processors performs $2p$ bus transactions for processors to obtain the lock and increment the counter, two bus transactions for the last processor to reset the counter and write the release flag, and another $p - 1$ bus transactions to read the flag after it has been invalidated. Note that this is better than the traffic for even a test-and-test&set lock to be acquired by p processes because, in that case, each of the p releases causes an invalidation that results in $O(p)$ processes trying to perform the test&set again, thus resulting in $O(p^2)$ bus transactions. However, the contention resulting from these competing bus transactions can be substantial if many processors arrive at the barrier simultaneously, so barriers can be expensive.

Improving Barrier Algorithms for a Bus

One part of the problem in the centralized barrier is that all processors contend for the same lock and flag variables. To address this, we can construct barriers that cause fewer processors to contend for the same variable. For example, processors can signal their arrival at the barrier through a software combining tree (see Section 3.3.2). In a binary combining tree, for example, only two processors notify each other of their arrival at each node of the tree, and only one of the two moves up to participate at the next higher level of the tree. Thus, only two processors ever access a given variable. In a distributed network with multiple parallel paths, such as those found in scalable machines, a combining tree can perform much better than a centralized barrier since different pairs of processors can communicate with each other

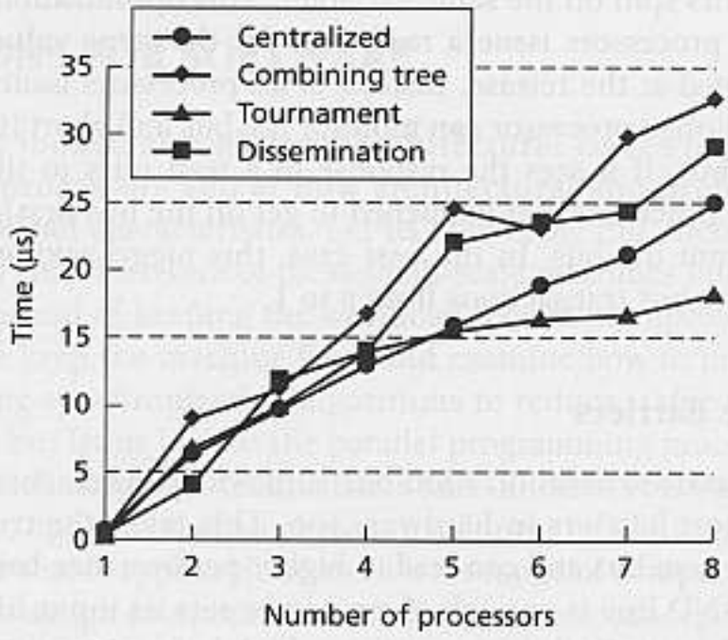

FIGURE 5.31 Performance of some barriers on the SGI Challenge. Performance is measured as average time per barrier over a loop of many consecutive barriers (with no work or delays between them). The higher critical path latency of the combining tree barrier hurts it on a bus, where it has no traffic and contention advantages.

in different parts of the network in parallel. However, with a centralized interconnect like a bus, even though pairs of processors communicate through different variables, they all generate bus transactions and hence serialization and contention on the same bus. Since a binary tree with p leaves has approximately $2p$ nodes, a combining tree requires a similar total number of bus transactions to the centralized barrier. It also has higher latency since, while it too requires $O(p)$ serialized bus transactions in all, even without bus serialization each processor must wait at least $\log p$ steps to get from the leaves to the root of the tree, each with significant work. The advantage of a combining tree for a bus is that it does not use locks but, rather, simple read and write operations, which may compensate for its larger uncontended latency if the number of processors on the bus is large. However, the simple centralized barrier performs quite well on a bus, as shown in Figure 5.31. Some of the other barriers shown in the figure for illustration will be discussed along with tree barriers in the context of scalable machines in Chapter 7.

Hardware Primitives

Since the centralized barrier uses locks and ordinary reads and writes, the hardware primitives needed depend on which lock algorithms are used. If a machine does not support atomic primitives well, combining tree barriers can be useful for bus-based machines as well.

A special bus primitive can be used to reduce the number of bus transactions for read misses in the centralized barrier (as well as for highly contended locks in which

processors spin on the same variable). This optimization takes advantage of the fact that all processors issue a read miss for the same value of the flag when they are invalidated at the release. Instead of all processors issuing a separate read-miss bus transaction, a processor can monitor the bus and abort its read miss before putting it on the bus, if it sees the response to a read miss to the same location (issued by another processor that happened to get on the bus first), and simply take the return value from the bus. In the best case, this piggybacking can reduce the number of read-miss bus transactions from p to 1.

Hardware Barriers

If a separate synchronization bus is provided, as discussed for locks, it can be used to support barriers in hardware too. This takes the traffic and contention off the main system bus and can lead to higher-performance barriers. Conceptually, a single wired-AND line is enough. A processor sets its input high when it reaches the barrier and waits until the output goes high before it can proceed. (In practice, reusing barriers requires that more than a single wire be used.) Such a separate hardware mechanism for barriers can be particularly useful if the frequency of barriers is very high, as it may be in programs that are automatically parallelized by compilers at the inner loop level and that need global synchronization after every innermost loop. However, its value in practice is unclear, and it can be difficult to manage when only a portion of the processors on the machine participate in the barrier. For example, it is difficult to dynamically change the number of processors participating in the barrier or to adapt the configuration of participating processors when processes are migrated among processors by the operating system. Having multiple participating processes running on the same processor also causes complications. Current bus-based multiprocessors therefore do not tend to provide special hardware support but build barriers in software out of locks and shared variables.

5.5.6 Synchronization Summary

Some bus-based machines have provided full hardware support for synchronization operations such as locks and barriers. However, concerns about flexibility have led most contemporary designers to provide support for only simple atomic operations in hardware and to synthesize higher-level synchronization operations from them in software libraries. The application programmer generally uses the libraries and can be unaware of the low-level atomic operations supported on the machine. The atomic operations may be implemented either as single instructions or through speculative read-write instruction pairs like load-locked and store-conditional. The greater flexibility of the latter is making them increasingly popular. We have already seen some of the interplay between synchronization primitives, algorithms, and architectural details. This interplay will be much more pronounced when we discuss synchronization for scalable shared address space machines in the coming chapters.

5.6 IMPLICATIONS FOR SOFTWARE

So far, we have looked at high-level architectural issues for bus-based cache-coherent multiprocessors and at how architectural and protocol trade-offs are affected by workload characteristics. Let us now come full circle and examine how the architectural characteristics of these small-scale machines influence parallel software. That is, instead of keeping the workload fixed and improving the machine or its protocols, we keep the machine fixed and examine how to improve parallel programs. Improving synchronization algorithms to reduce traffic and latency was an example of this, but let us look at the parallel programming process more generally.

The general techniques for load balance and inherent communication discussed in Chapter 3 also apply to cache-coherent machines. In addition, one general partitioning principle that is applicable across a wide range of computations on these machines is to try to assign computation such that only one processor writes a given set of data, at least during a single computational phase. In many computations, processors read one large shared data structure and write another. In Raytrace, for example, processors read a scene and write an image. A choice is available of whether to partition the computation so the processors write disjoint pieces of the destination structure and read share the source structure, or read disjoint pieces of the source structure and write share the same memory locations in the destination. All other considerations being equal (such as load balance and programming complexity), it is usually advisable to avoid write sharing in these situations. Write sharing not only causes invalidations and, hence, cache misses and traffic, but if different processes write the same words, it is very likely that the writes must be protected by synchronization such as locks, which are even more expensive.

The structure of communication is not much of a variable: with a single centralized memory, little incentive exists to use explicit memory-to-memory data transfers, so all communication is implicit through loads and stores that lead to the transfer of cache blocks. Mapping is not an issue (other than to try to ensure that processes migrate from one processor to another as little as possible) and is invariably left to the operating system. The most interesting issues are managing data locality and artifactual communication in the orchestration step, and in particular, addressing temporal and spatial locality to reduce the number of cache misses and hence reduce latency, traffic, and contention on the shared bus.

With main memory being centralized, temporal locality is exploited in the processor caches. The specialization of the working set curve introduced in Chapter 3 for bus-based machines is shown in Figure 5.32. All capacity-related misses go to the same bus and memory and are about as expensive as coherence misses. The other three kinds of misses will occur and generate bus traffic even with an infinite cache. The major goal for temporal locality is to have working sets fit in the cache hierarchy, and the techniques are the same as those discussed in Chapter 3.

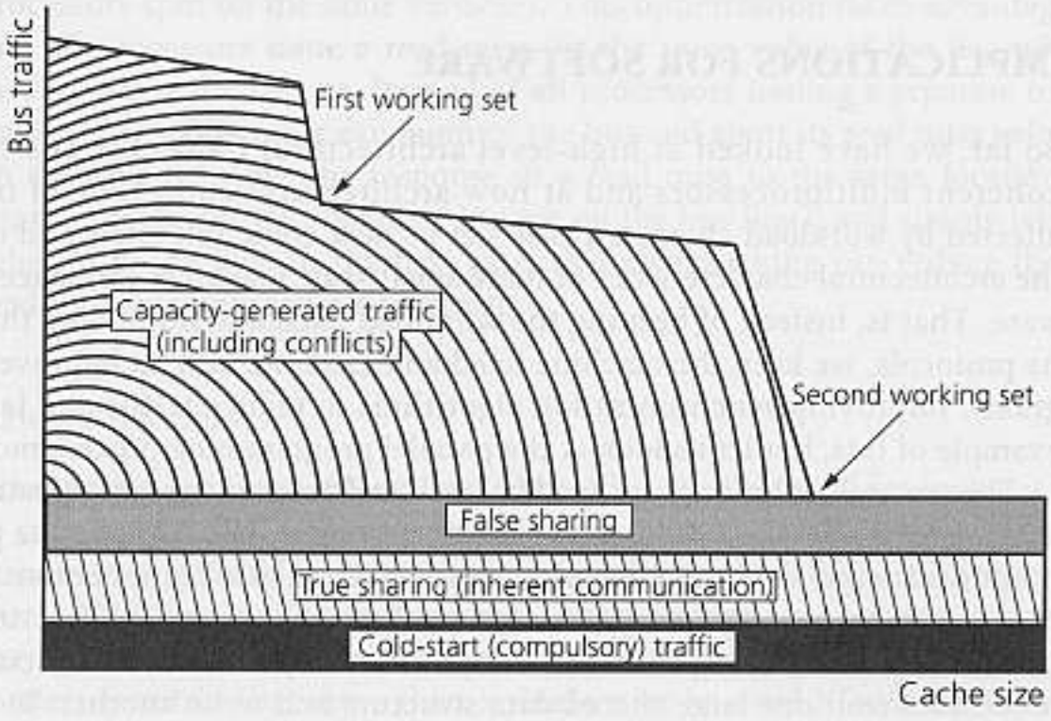

FIGURE 5.32 Data traffic on the shared bus and its components as a function of cache size. The points of inflection indicate the working sets of the program.

For spatial locality, a centralized memory makes data distribution and the granularity of allocation in main memory irrelevant (only interleaving data among memory banks to reduce contention may be an issue, just as in uniprocessors). The ill effects of poor spatial locality are *fragmentation* (i.e., fetching unnecessary data on a cache block) and false sharing. The reasons are that the granularity of communication and the granularity of coherence are both cache blocks, which are larger than a word. The former causes fragmentation, and the latter causes false sharing. (We assume here that techniques to eliminate false sharing like subblock dirty bits are not used since they are not found in most real machines.) Let us examine some techniques to alleviate these problems and effectively exploit the prefetching effects of long cache blocks, as well as techniques to alleviate cache conflicts by better spatial organization of data. Many such techniques can be found in a programmer's "bag of tricks." The following provides only a sampling of the most general ones.

- *Assign tasks to reduce spatial interleaving of access patterns.* It is desirable to assign tasks such that each processor tends to access large contiguous chunks of data. For example, if an array computation with n elements is to be divided among p processors, it is better to divide it so that each processor accesses n/p contiguous elements rather than to use a finely interleaved assignment of elements. This increases spatial locality and reduces false sharing of cache blocks. Of course, load balancing or other constraints may force us to do otherwise.
- *Structure data to reduce spatial interleaving of access patterns.* We saw an example of this in the equation solver kernel in Chapter 3, when we used higher-dimensional arrays to keep a processor's partition of an array contiguous in the

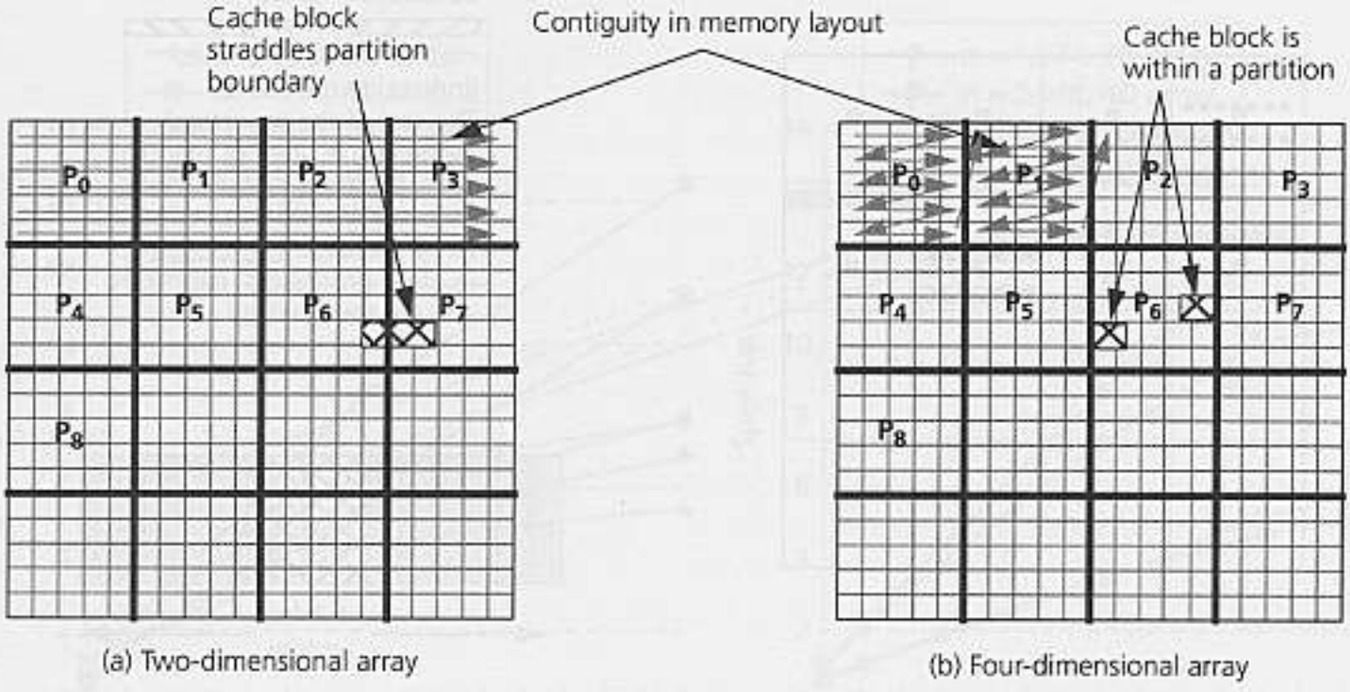

FIGURE 5.33 Reducing false sharing and fragmentation by using higher-dimensional arrays to keep partitions contiguous in the address space. In the two-dimensional array case, cache blocks straddling partition boundaries cause both fragmentation (a miss brings in useless data from the other processor's partition) as well as false sharing. The four-dimensional array representation makes partitions contiguous and alleviates these problems.

address space in order to allocate partitions locally at page granularity in physically distributed memory. This technique also helps reduce false sharing, fragmentation of data transfer, and conflict misses, as shown in Figures 5.33 and 5.34, all of which cause misses and traffic on the bus. A cache block larger than a single grid element may straddle a column-oriented partition boundary, as shown in Figure 5.33(a). If the block is larger than two grid elements, it can cause communication due to false sharing. This is easiest to see if we assume for a moment that there is no inherent communication in the algorithm; for example, suppose in each sweep a process simply adds a constant value to each of its assigned grid elements instead of performing a nearest-neighbor computation. Now, even a two-element (or larger) cache block straddling a partition boundary would be false-shared as different processors wrote different words on it. This would also cause fragmentation in communication, since a process reading its own boundary element and missing on it would also fetch other elements in the other processor's partition that are on the same cache block but that it does not need. The conflict-misses problem is explained in Figure 5.34. The issue in all these cases is noncontiguity of partitions. Thus, a single data structure transformation (as in Figure 5.33[b]) helps us solve all our spatial locality–related problems in the equation solver kernel. Figure 5.35 illustrates the performance impact of using higher-dimensional arrays to represent grids or blocked matrices in the Ocean and LU applications on the SGI

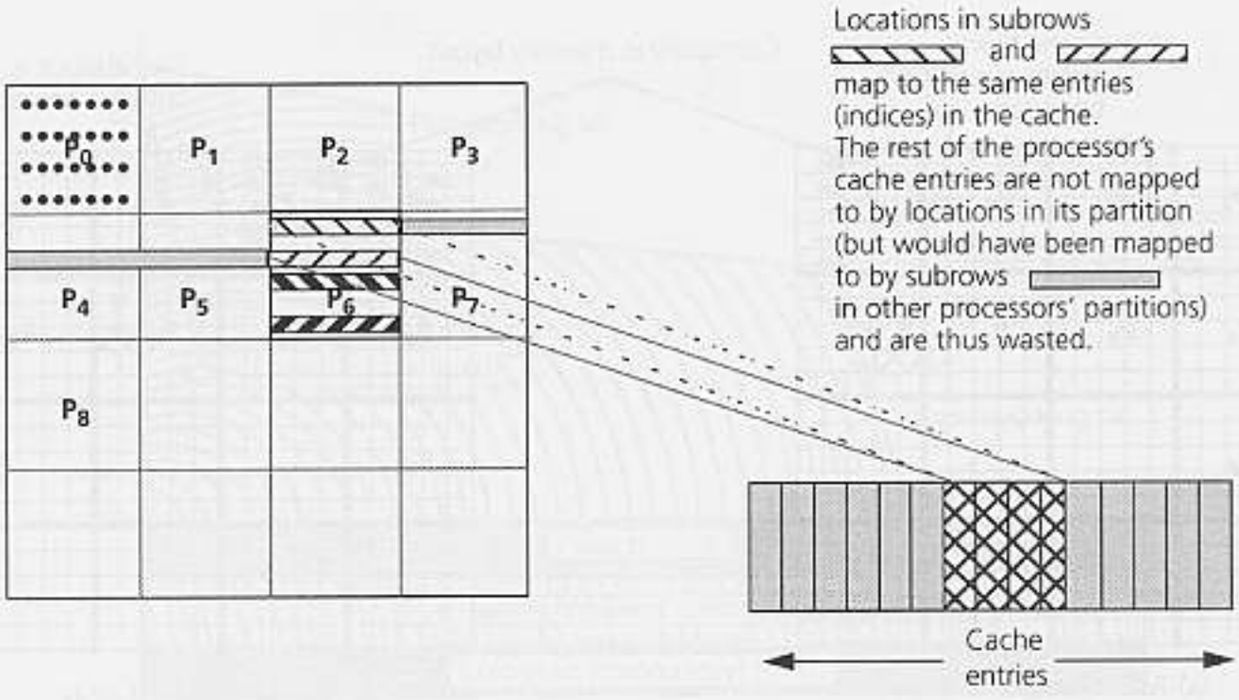

FIGURE 5.34 Cache mapping conflicts caused by a two-dimensional array representation in a direct-mapped cache. The figure shows the worst case, in which the separation between successive subrows in a process's partition (i.e., the size of a full row of the 2D array) is exactly equal to the size of the cache, so consecutive subrows map directly on top of one another in the cache. Every subrow accessed knocks the previous subrow out of the cache. In the next sweep over its partition, the processor will miss on every cache block it references, even if the cache as a whole is large enough to fit a whole partition. Many intermediately poor cases may be encountered depending on grid size, number of processors, and cache size. Since the cache size in bytes is a power of two, sizing the dimensions of allocated arrays to be powers of two is discouraged.

Challenge. The impact of conflicts and false sharing on uniprocessor and multiprocessor performance is clear.

- *Beware of conflict misses.* In illustrating conflict misses in the grid solver, Figure 5.34 shows how allocating power-of-two-sized arrays can cause pathological cache conflict problems since the cache size is also a power of two. Even if the logical size of the array that the application needs is a power of two, it is often useful to allocate a larger array that is not a power of two and then access only the amount needed. However, this strategy can interfere with allocating data at page granularity (also a power of two) in machines with physically distributed memory, so we may have to be careful. The cache mapping conflicts in this example are within a single data structure that is accessed in a predictable manner and can thus be alleviated in a structured way. Mapping conflicts are more difficult to avoid when they happen across different major data structures (e.g., across different grids used by the Ocean application), where they may have to be alleviated by ad hoc padding and alignment. However, in a shared address space they are particularly insidious when they occur on seemingly harmless shared variables or data structures that a programmer is not inclined to think about. For example, a frequently accessed pointer to an important data structure may conflict in a direct-mapped cache

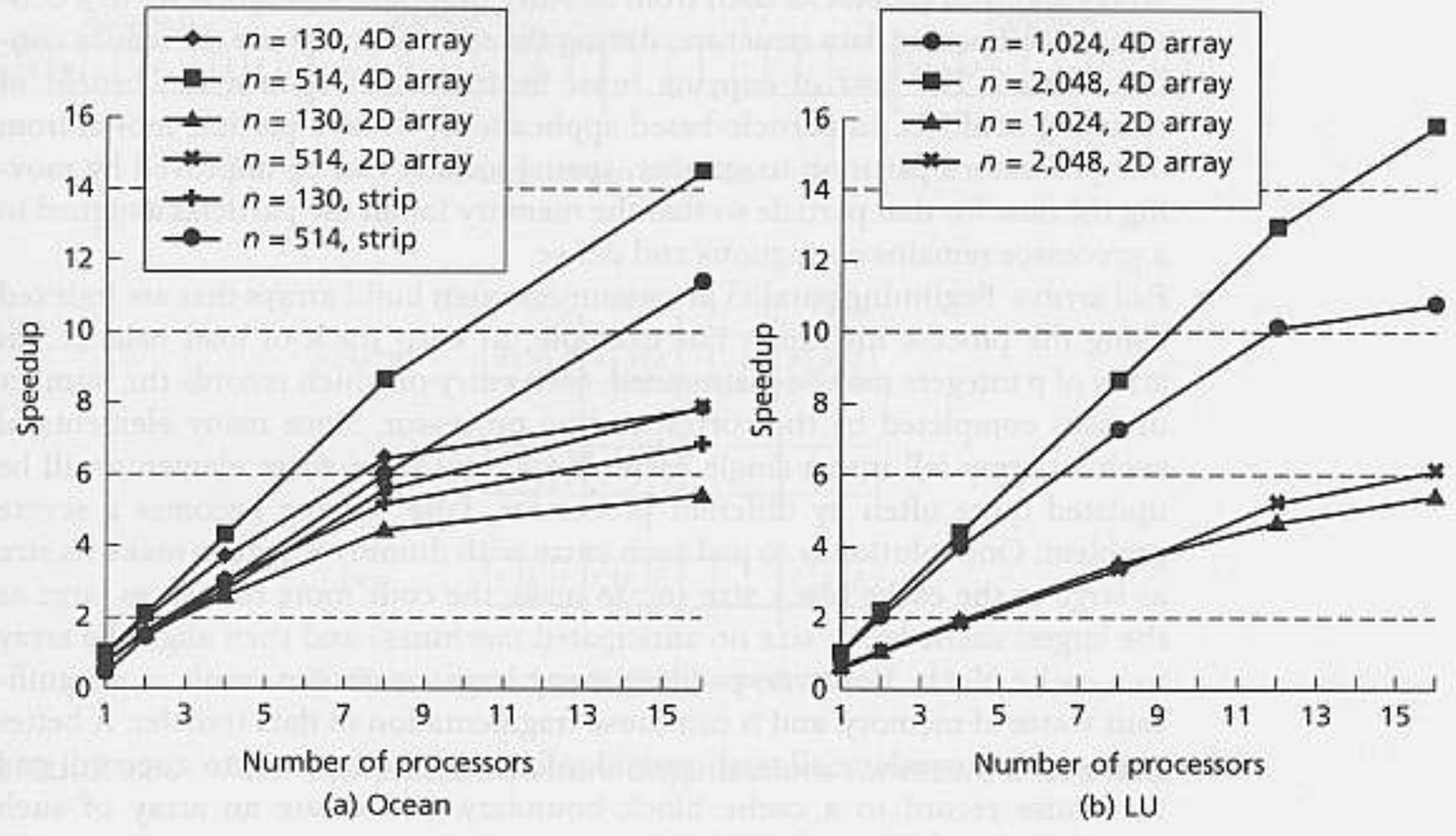

FIGURE 5.35 Performance impact of using 4D versus 2D arrays to represent two-dimensional grid or matrix data structures on the SGI Challenge. Results are shown for different problem sizes for the Ocean and LU applications. For Ocean, "strip" indicates partitioning into strips of contiguous rows (in which 2D or 4D arrays don't matter), while all other cases assume partitioning into squarelike blocks.

with a scalar variable that is also frequently accessed during the same computation, causing a lot of traffic. Fortunately, such problems tend to be infrequent in modern (large and set-associative) second-level caches. In general, efforts to exploit locality can be wasted if attention is not paid to reducing conflict misses.

- *Use per-processor heaps.* It is desirable to have separate heap regions for each processor (or process) from which it allocates data dynamically. Otherwise, if a program performs a lot of very small memory allocations, data used by different processors may fall on the same cache block.
- *Copy data to increase spatial locality.* If a processor is going to reuse a set of data that is otherwise allocated noncontiguously in the address space, it is often desirable to make a contiguous copy of the data for that period to improve spatial locality and reduce cache conflicts. Copying requires memory accesses and has a cost, and it is not useful if the data is likely to reside in the cache anyway. For example, in blocked matrix factorization or multiplication, with a 2D array representation of the matrix a block is not contiguous in the address space (just like a partition in the equation solver kernel). However, a 2D representation makes programming easier. It is therefore not uncommon to use 2D

arrays and to copy blocks used from another processor's assigned set to a contiguous temporary data structure, during the time of active use, to reduce conflict misses. The cost of copying must be traded off against the benefit of reducing conflicts. In particle-based applications, when a particle moves from one processor's partition to another, spatial locality can be improved by moving the data for that particle so that the memory for all the particles assigned to a processor remains contiguous and dense.

- *Pad arrays.* Beginning parallel programmers often build arrays that are indexed using the process identifier. For example, to keep track of load balance, an array of p integers may be maintained, each entry of which records the number of tasks completed by the corresponding processor. Since many elements of such an array fall into a single cache block, and since these elements will be updated quite often by different processors, false sharing becomes a severe problem. One solution is to pad each entry with dummy words to make its size as large as the cache block size (or, to make the code more robust, as large as the largest cache block size on anticipated machines) and then align the array to a cache block. However, padding many large arrays can result in a significant waste of memory, and it can cause fragmentation in data transfer. A better strategy is to combine all such variables for a given process into a record, pad the entire record to a cache block boundary, and create an array of such records indexed by process identifier.
- *Determine how to organize arrays of records.* Suppose we have a number of logical records to represent, such as the particles in the Barnes-Hut gravitational simulation. Should we represent them as a single array of n particles, each entry being a record with fields like position, velocity, force, mass, and so on, as in Figure 5.36(a)? Or should we represent them as separate arrays of size n, one per field, as in Figure 5.36(b)? Programs written for vector machines such as traditional CRAY computers tend to use a separate array (vector) for each property or field of an object—in fact, even one per field per physical dimension (x, y, or z). When data is accessed by field, for example, the velocity of all particles, this increases the performance of vector operations by making accesses to memory unit stride and hence reducing memory bank conflicts. In cache-coherent multiprocessors, however, new trade-offs arise, and the best way to organize data depends on the access patterns.

  An interesting tension is illustrated by the particle update and force calculation phases of the Barnes-Hut application. Consider the update phase first. A processor reads and writes only the position and velocity fields of all its assigned particles in this phase. However, its assigned particles are not contiguous in the shared particle array. Suppose there is one array of size n (number of particles) per field or property. A double-precision three-dimensional position (or velocity) is 24 bytes of data, so several of these may fit on a cache block. Since adjacent particles in the array may be read and written by different processors, false sharing can result. For this phase, it is better to have a single array of particle records, where each record holds all information about that particle; that is, to organize data by particle rather than by field.

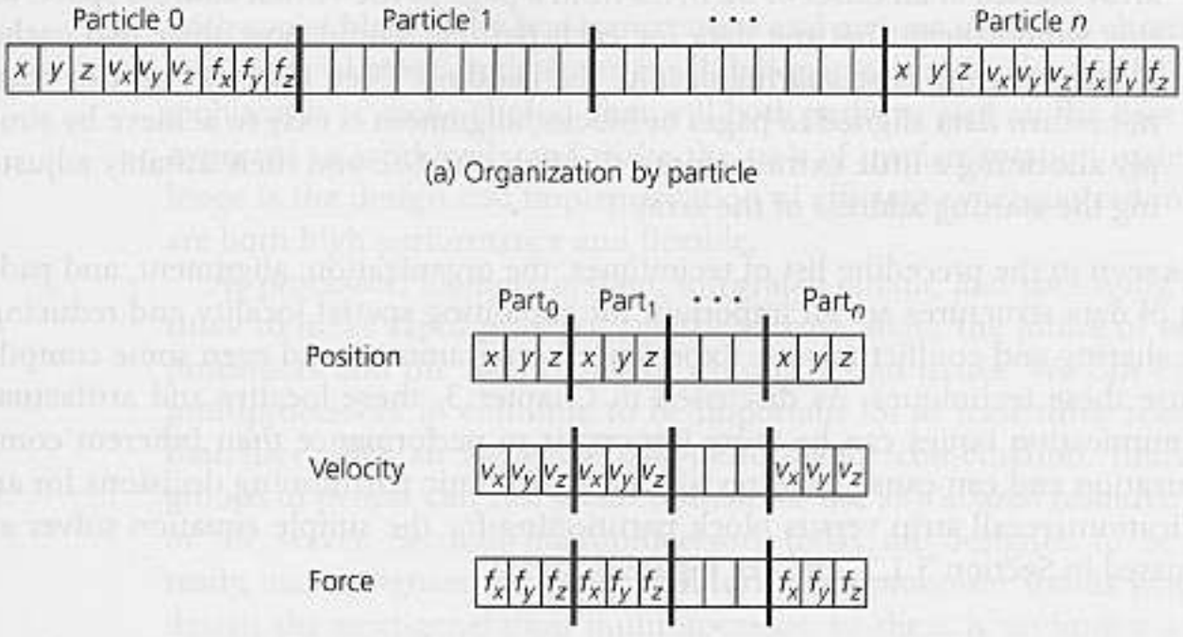

FIGURE 5.36 Alternative data structure organizations for record-based data

Now consider the force calculation phase of the same application. Suppose we use an organization by particle rather than by field as above. To compute the force on a particle, a processor reads the position values of many other particles and cells; it then updates the force components of its own particle. However, the force and position components of a particle may fall on the same cache block. In updating force components, it may therefore invalidate the position values of this particle from the caches of other processors that are using and reusing them as a result of false sharing within a particle record, even though the position values themselves are not being modified in this phase of computation. In this case, it would probably be better if we were to split the single array of particle records into two arrays of size n each, one for positions (and perhaps other properties) and one for forces. The entries of the force array themselves could be padded to reduce cross-particle false sharing. In general, it is often beneficial to split arrays of records to separate fields that are used in a read-only manner in a phase from the fields whose values are updated in the same phase. Different situations or phases may dictate different organizations for a data structure, and the ultimate decision depends on which pattern or phase dominates performance.

- *Align arrays.* In conjunction with the preceding techniques, it is often necessary to align arrays to cache block boundaries to achieve the full benefits. For example, given a cache block size of 64 bytes and 8-byte fields, we may have decided to maintain a single array of particle records with x, y, z, fx, fy, and fz. To avoid cross-particle false sharing, we pad each 48-byte record with two dummy 8-byte fields to fill a cache block. However, this wouldn't help if the

array started at an offset of 32 bytes from a page in the virtual address space, as this would mean that the data for each particle would now span two cache blocks, causing false sharing despite the padding. Even if a `malloc` call does not return data aligned to pages or blocks, alignment is easy to achieve by simply allocating a little extra memory through `malloc` and then suitably adjusting the starting address of the array.

As seen in the preceding list of techniques, the organization, alignment, and padding of data structures are all important for exploiting spatial locality and reducing false sharing and conflict misses. Experienced programmers and even some compilers use these techniques. As discussed in Chapter 3, these locality and artifactual communication issues can be more important to performance than inherent communication and can cause us to revisit our algorithmic partitioning decisions for an application (recall strip versus block partitioning for the simple equation solver as discussed in Section 3.1.2, and see Figure 5.35[a]).

5.7 CONCLUDING REMARKS

Symmetric shared memory multiprocessors are a natural extension of workstations and personal computers. A sequential application can run totally unchanged and yet benefit in performance by obtaining a larger fraction of a processor's time and by taking advantage of the large amount of shared main memory and I/O capacity typically available on such machines. Parallel applications are also relatively easy to bring up, as all shared data is directly accessible from all processors using ordinary loads and stores. Gradual parallelization is possible by selectively parallelizing computationally intensive portions of a sequential application, subject to the dictates of Amdahl's Law. For multiprogrammed workloads, a key advantage is the fine granularity at which resources can be shared among application processes and by the operating system, which can thus easily export a familiar, single-system image to each application. This is true both temporally, in that processors and/or main memory pages can frequently be reallocated among different application processes, and physically, in that main memory may be split among applications at the granularity of individual pages. Because of these appealing features, all major vendors of computer systems, from workstation suppliers like Sun, Silicon Graphics, Hewlett-Packard, Digital, and IBM to personal computer suppliers like Intel and Compaq, are producing and selling such machines. In fact, for some of the large workstation vendors, these multiprocessors constitute a substantial fraction of their revenue stream and a still larger fraction of their net profits because of the higher margins on these higher-end machines.

The key technical challenge in the design of symmetric multiprocessors is the organization and implementation of the shared memory system, which is used for communication between processors in addition to handling all regular memory accesses. Most small-scale parallel machines found today use the system bus as the interconnect for communication, and the challenge then becomes how to maintain coherency of the shared data in the private caches of the processors. A large variety

of options are available to the system architect, including the set of states associated with cache blocks, the bus transactions and actions used, the choice of cache block size, and whether updates or invalidations are used. The key task of the system architect is to make choices that will both perform well on the data sharing patterns expected in workloads and make the task of implementation easier. Another challenge is the design and implementation of efficient synchronization techniques that are both high performance and flexible.

As processor, memory system, integrated circuit, and packaging technology continue to make rapid progress, questions arise about the future of small-scale multiprocessors and the importance of various design issues. We can expect small-scale multiprocessors to continue to be important for at least three reasons. The first is that they offer an attractive cost-performance combination. Individuals or small groups of people can easily afford them for use as a shared resource or as a compute or file server. Second, microprocessors today are designed to be multiprocessor-ready, and designers are aware of future microprocessor trends when they begin to design the next-generation multiprocessor, so there is no longer a significant time lag between the latest microprocessor and its incorporation in a multiprocessor. As we saw in Chapter 1, the Intel Pentium Pro processor line plugs "gluelessly" into a shared bus. The third reason is that the essential software technology for parallel machines (compilers, operating systems, programming languages) is maturing rapidly for small-scale shared memory machines. For example, most computer system vendors have efficient parallel versions of their operating systems ready for their bus-based multiprocessors. As levels of integration increase, multiple processors on a chip become attractive. While the optimal design points may change, the design issues that we have explored in this chapter are fundamental and will remain important with progress in technology.

This chapter has explored many of the key design aspects of bus-based multiprocessors at the "logical" level, involving cache block state transitions and complete (atomic) bus transactions. At this level, the design and implementation appears to be a rather simple extension of traditional cache controllers. However, much of the difficulty in such designs and many of the opportunities for optimization and innovation occur at the next lower level of protocol design and at the more detailed "physical" level. The next chapter goes down a level deeper into the design and organization of bus-based cache-coherent multiprocessors and some of their natural generalizations.

5.8 EXERCISES

5.1 Is the cache coherence problem an issue with processor registers? Given that registers are not kept consistent in hardware, how do current systems guarantee the desired semantics of a program?

5.2 Consider the following graph indicating the miss rate of an application as a function of cache block size on a multiprocessor. As might be expected, the curve has a U-shaped appearance. Consider the three points A, B, and C on the curve. Indicate