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

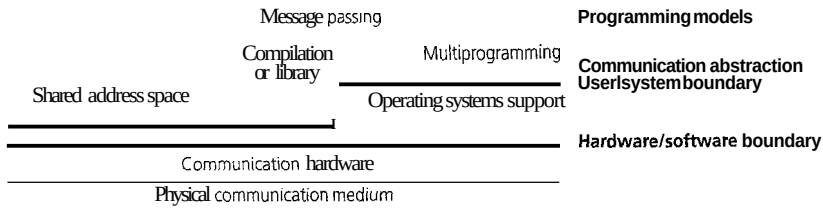

FIGURE 5.1 Layers of abstraction of the communication architecture for bus-based SMPs. A shared address space is supported directly in hardware, while message passing is supported in software

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

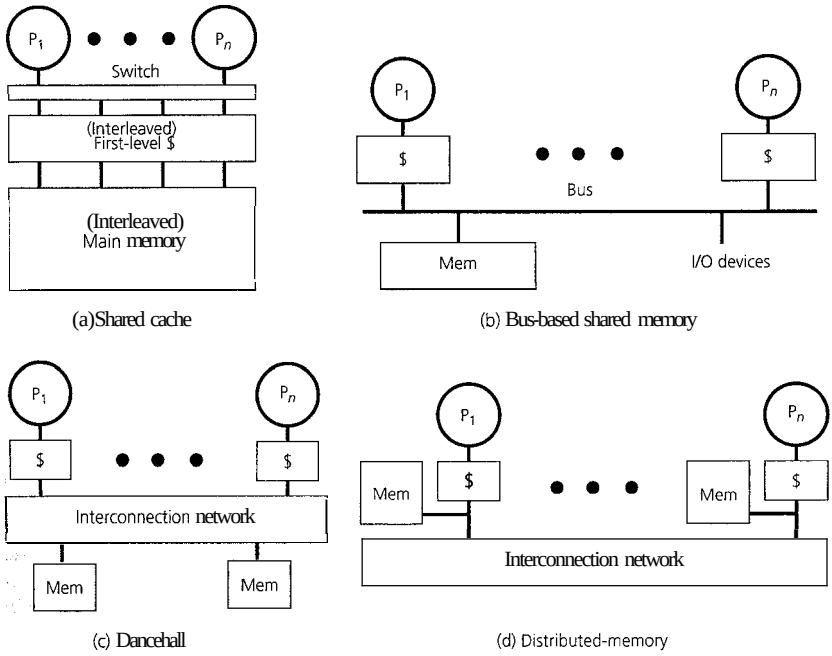

FIGURE 5.2 Common extended memory hierarchies found in multiprocessors

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 indeedfar 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 ıs reduced because the data accesses issued by a processor that are satisfied ın the cache do not have to appear on the lnterconnect In all but the shared cache approach, each processor has at least one level of ıts cache hierarchy that ıs pnvate This raises a critical challenge—namely, that of *cache* coherence The problem anses when copies of the same memory block are present ın the caches of one or more processors, if a processor writes to and hence modifies that memory block, then, unless special actıon ıs taken, the other processors will continue to access the old, stale copy of the block that ıs in their caches

Currently, most small-scale multiprocessors use a shared bus lnterconnect 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 settıngs Specific organızatıons may change as technology evolves However, besides beıng the most popular, the bus-based and distributed-memory organızatıons also illustrate the two fundamental approaches to solving the cache coherence problem, dependmg on the nature of the Interconnect one for the case where any transactıon placed on the Interconnect ıs visible to all processors (like a bus) and the other where the lnterconnect is decentrahzed and a point-to-point transactıon is visible only to the processors at ıts endpoints This chapter focuses on the logical design of protocols that exploit the fundamental propertıes of a bus to solve the cache coherence problem The next chapter expands on the design issues associated with realizing these cache coherence techniques ın hardware The basic design of scalable distributed-memory mult~processors will be addressed ın Chapter 7, followed by coverage of the ıssues specific to scalable cache coherence ın Chapters 8 and 9

Section 5 1 descnbes the cache coherence problem for shared memory architectures ın detail and descnbes the simplest example of what are called *snooping* cache coherence protocols Coherence ıs not only a key hardware design concept but ıs a necessary part of our intulhve notion of the abstraction of memory. However, parallel software often makes stronger assumptıons than coherence about how memory behaves Section 5 2 extends the discussion of ordenng begun ın Chapter 1 and introduces the concept of memory consıstency, which defines the semantıcs of shared address space This ıssue has become increasingly Important ın computer architecture and compiler design, a large fraction of the reference manuals for most recent ınstruction set architectures ıs devoted to the memory consistency model Once the abstractions and concepts are defined, Section 5 3 presents the design space for more reahstic snoopmg protocols and shows how they satisfy the conditions for coherence as well as for a useful consistency model. It describes the operadon 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

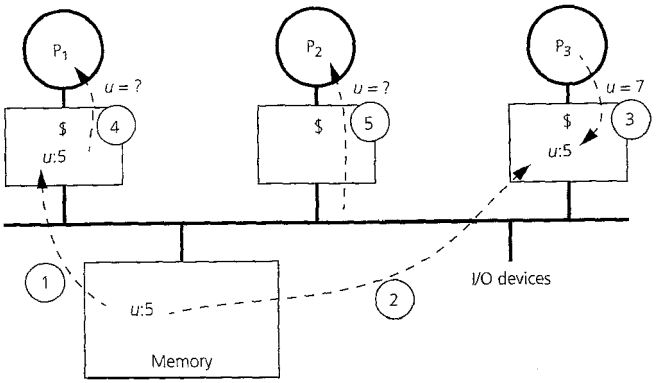

FIGURE 5.3 Example cache coherence **problem.** The figure shows three processors with caches connected by a bus to main memory. u is a location in memory whose contents are being read and written by the processors. The sequence in which reads and writes are done is indicated by the number listed inside 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 VO operations occur. Most VO 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

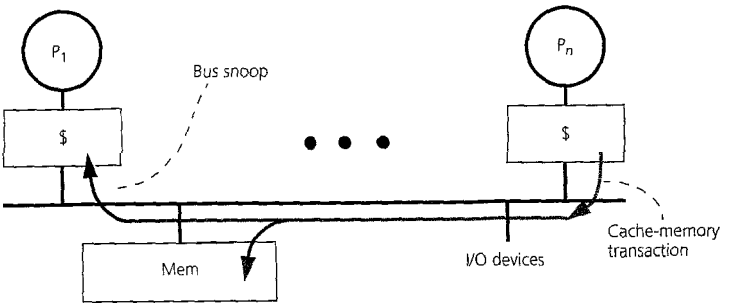

FIGURE 5.4 A snooping cache-coherent multiprocessor. Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously "snoops" on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent. The gray arrows 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-

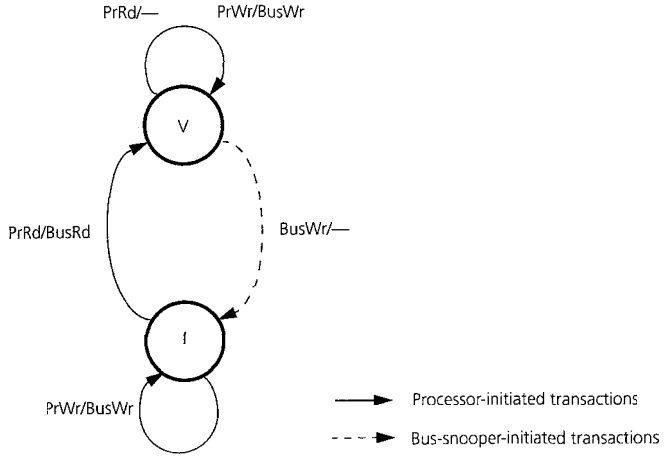

FIGURE 5.5 Snoopy coherence for a multiprocessor with write-through, write-no-allocate caches. There are two states, valid (V) and invalid (I), with intuitive semantics. The notation *A/B* (e.g., PrRd/BusRd) means if A is observed, then transaction *B* is generated. From the processor side, the requests 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?

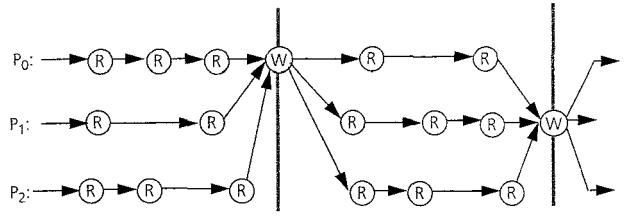

FIGURE 5.6 Partial order of memory operations for an execution with the write-through invalidation protocol. Write bus transactions define a global sequence of events between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving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

Answer A single processor will generate 30 million stores per second (0.15 stores per instruction x 1 instruction per cycle x 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 (hut 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, la, 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, la 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.

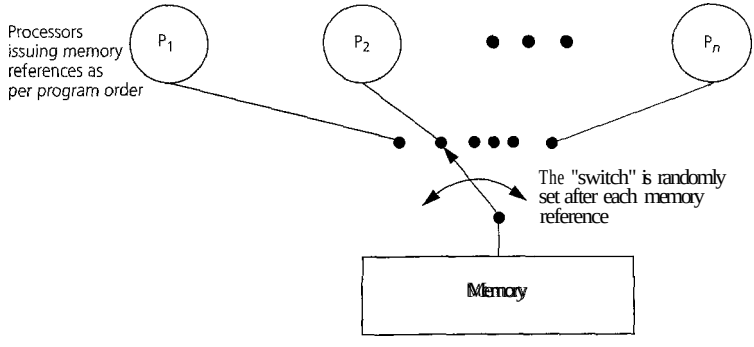

FIGURE 5.10 Programmer's abstraction of the memory subsystem under the sequential consistency model. The model completely hides the underlying concurrency in the memory system hardware (e.g., the possible existence of distributed main memory, the presence of caches and write buffers) from the programmer.

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.

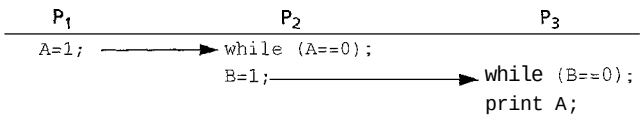

FIGURE 5.11 Example illustrating the importance of write atomicity for sequential consistency

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 sausfy 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 Bnggs 1986, Scheunch 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, modem 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 modem 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 machme, so the write propagation property of coherence is violated.

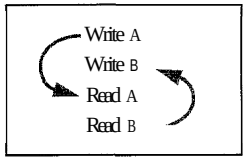

FIGURE 5.12 Preserving the orders in a sequential program running on a uniprocessor. Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

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 he 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

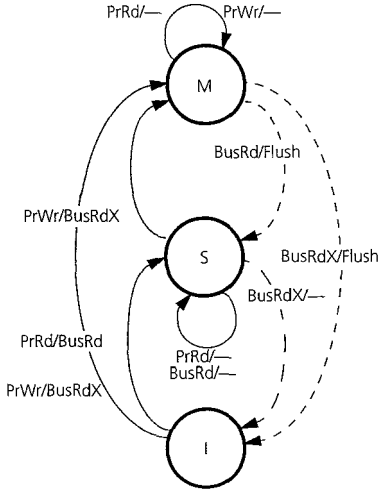

FIGURE 5.13 Basic three-state invalidation protocol. M, S, and I stand for modified, shared, and invalid states, respectively. The notation *A/B* means that if the controller observes the event A from the processor side or the bus side, then in addition to the state change, it generates the bus transaction 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 afier 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($\tilde{S}$) 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-cachesharing. 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 modem 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

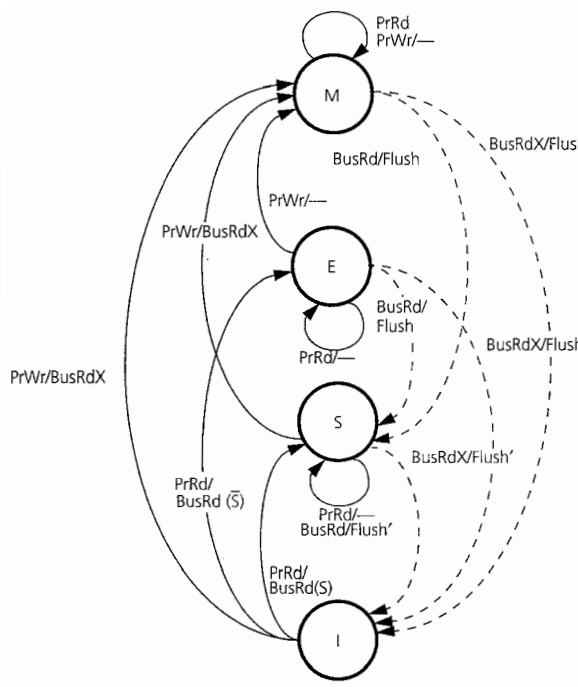

FIGURE 5.15 State transition diagram for the Illinois MESI protocol. MESI stands for the modified (dirty), exclusive, shared, and invalid states, respectively. The notation is the same as that in Figure 5.13. The Estate helps reduce bus traffic for sequential programs where data is not shared. Whenever 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 mole 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 m Sc state Or it may be that no cache has it in Sm state, but some have it m 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 m 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 hit 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.

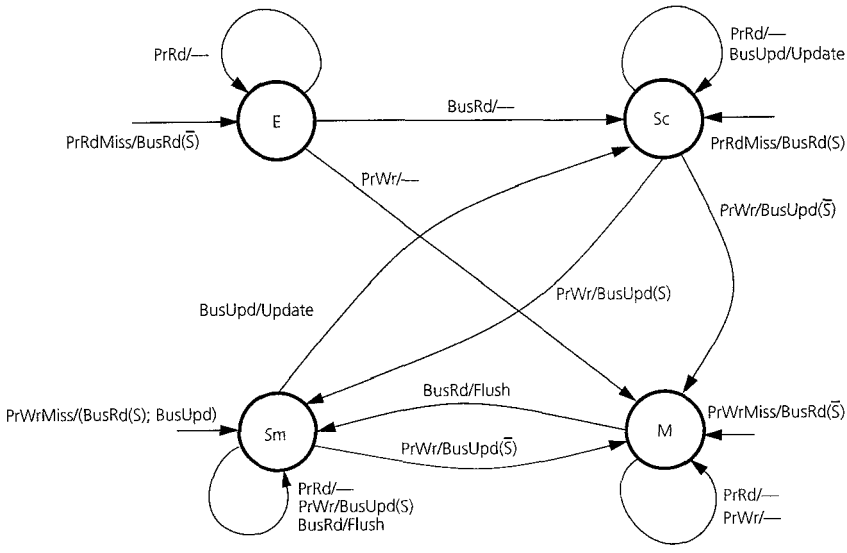

FIGURE 5.16 State transition diagram for the Dragon update protocol. The four states are exclusive (E), shared-clean (Sc), shared-modified (Sm), and modified (M). There is no invalid (I) state because the update protocol always keeps blocks in the cache up-to-date.

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 $\rightarrow$ 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	From	To: NP	I	E	S	M
Barnes-Hut	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	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	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	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	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 insuucuon 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

From \ To	NP	I	E	S	M
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

Gwen 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. ■