5

Shared Memory Multiprocessors

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

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

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

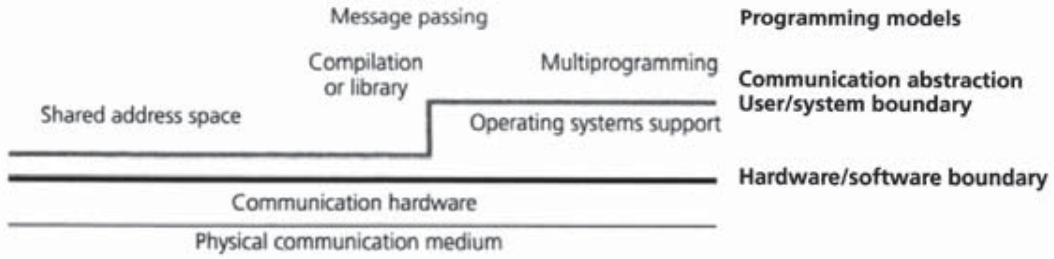

FIGURE 5.1 Layers of abstraction of the communication architecture for bus-based SMPs. A shared address space is supported directly in hardware, while message passing is supported in software.

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

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

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

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

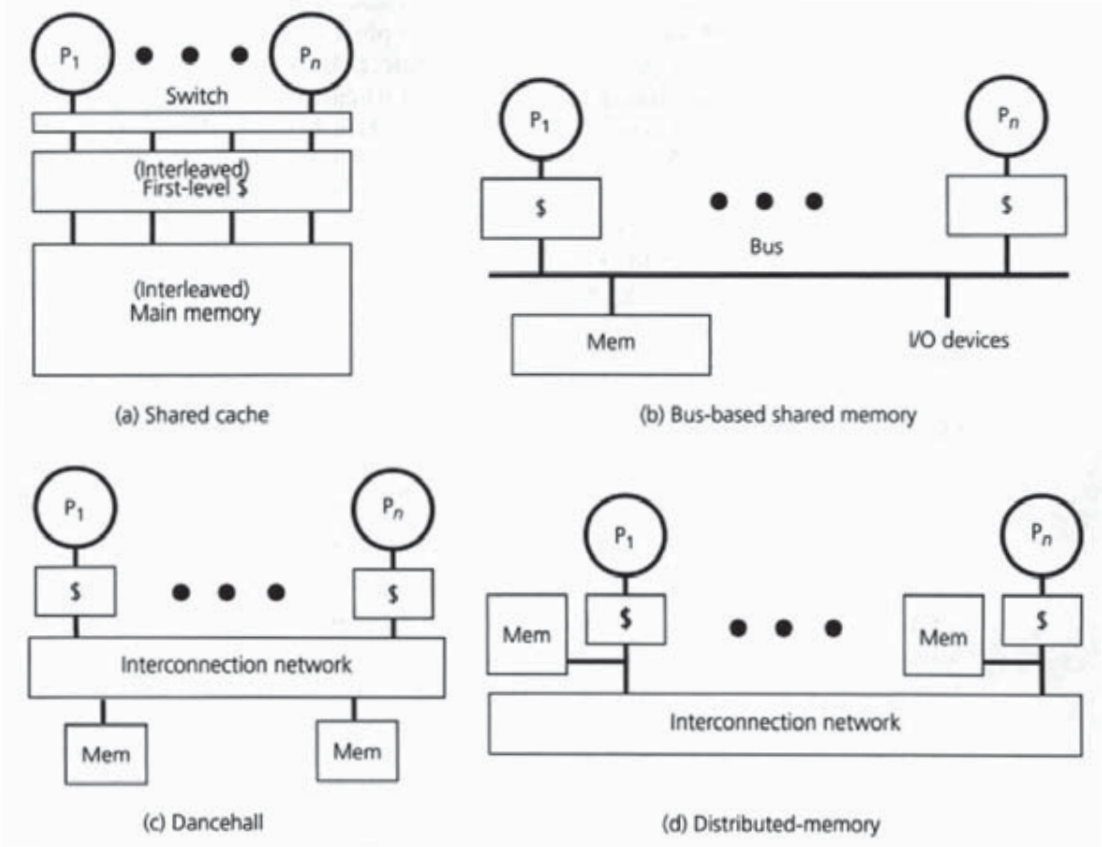

FIGURE 5.2 Common extended memory hierarchies found in multiprocessors

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

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

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

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

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

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

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

5.1 CACHE COHERENCE

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

5.1.1 The Cache Coherence Problem

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

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

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

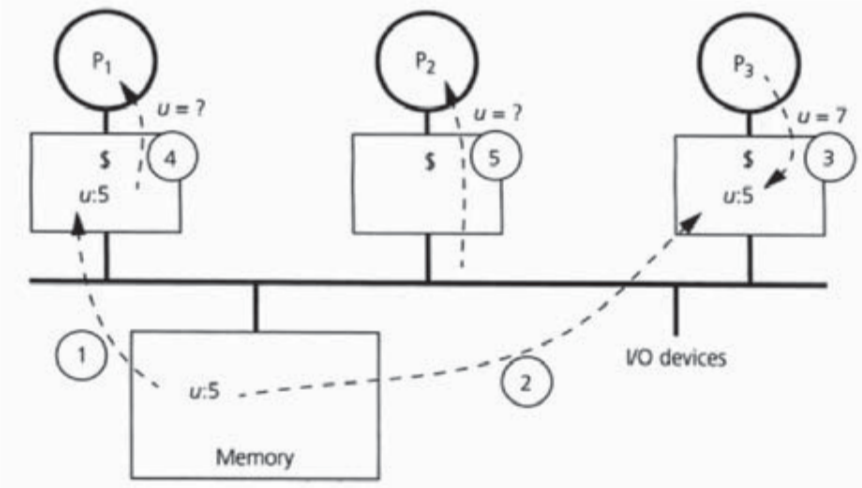

FIGURE 5.3 Example cache coherence problem. The figure shows three processors with caches connected by a bus to main memory. *u* is a location in memory whose contents are being read and written by the processors. The sequence in which reads and writes are done is indicated by the number listed inside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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.2 Cache Coherence through Bus Snooping

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

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

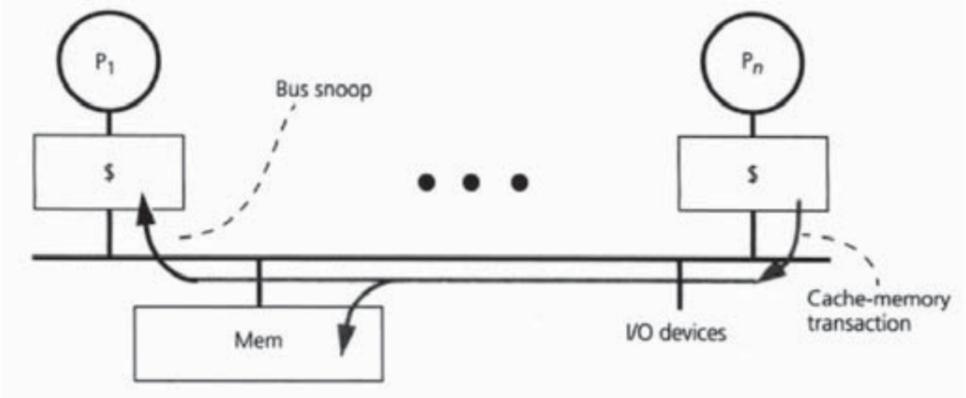

FIGURE 5.4 A snooping cache-coherent multiprocessor. Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously "snoops" on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent. The gray arrows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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

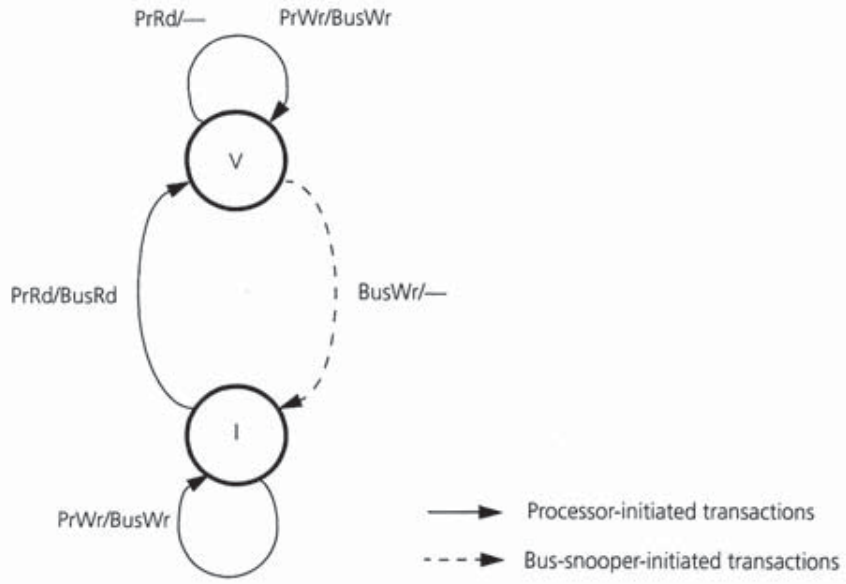

FIGURE 5.5 Snoopy coherence for a multiprocessor with write-through, write-no-allocate caches. There are two states, valid (V) and invalid (I), with intuitive semantics. The notation *A/B* (e.g., PrRd/BusRd) means if *A* is observed, then transaction *B* is generated. From the processor side, the requests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

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

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

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

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

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

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

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

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

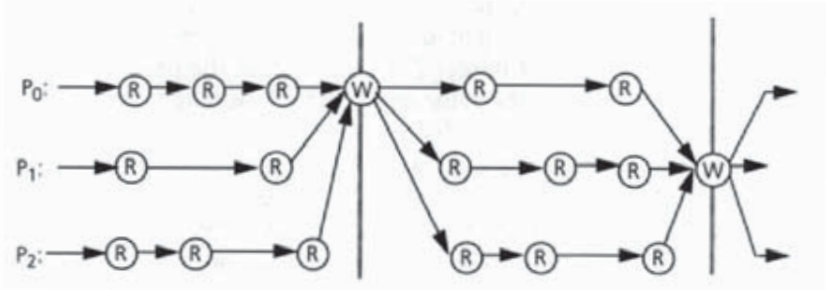

FIGURE 5.6 Partial order of memory operations for an execution with the write-through invalidation protocol. Write bus transactions define a global sequence of events between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving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

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

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

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

5.2 MEMORY CONSISTENCY

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