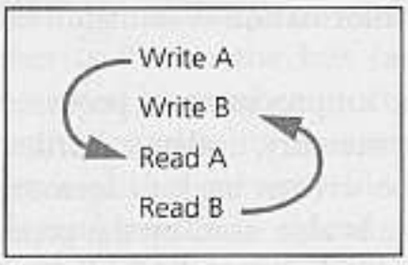

FIGURE 5.12 Preserving the orders in a sequential program running on a uniprocessor. Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

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

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

5.3 DESIGN SPACE FOR SNOOPING PROTOCOLS

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

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

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

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

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

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

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

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

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

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

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

We assume that the bus allows the following transactions:

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

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

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

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

State Transitions

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

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

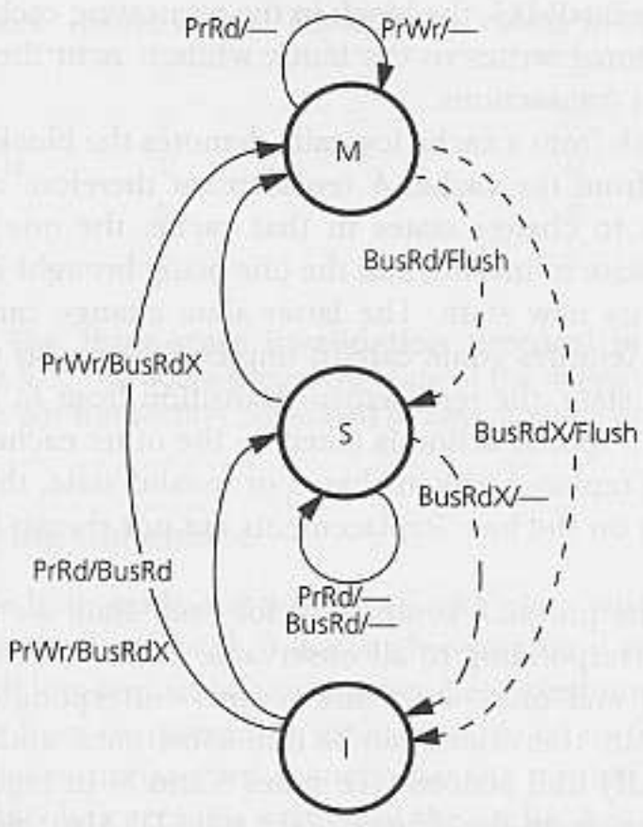

FIGURE 5.13 Basic three-state invalidation protocol. M, S, and I stand for modified, shared, and invalid states, respectively. The notation *A/B* means that if the controller observes the event *A* from the processor side or the bus side, then in addition to the state change, it generates the bus transaction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

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

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

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

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

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

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

Answer The results are shown in Figure 5.14. ■

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

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

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

Satisfying Coherence

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

Satisfying Sequential Consistency

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

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

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

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

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

Lower-Level Design Choices

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

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

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

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

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

State Transitions

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

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

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

Lower-Level Design Choices

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

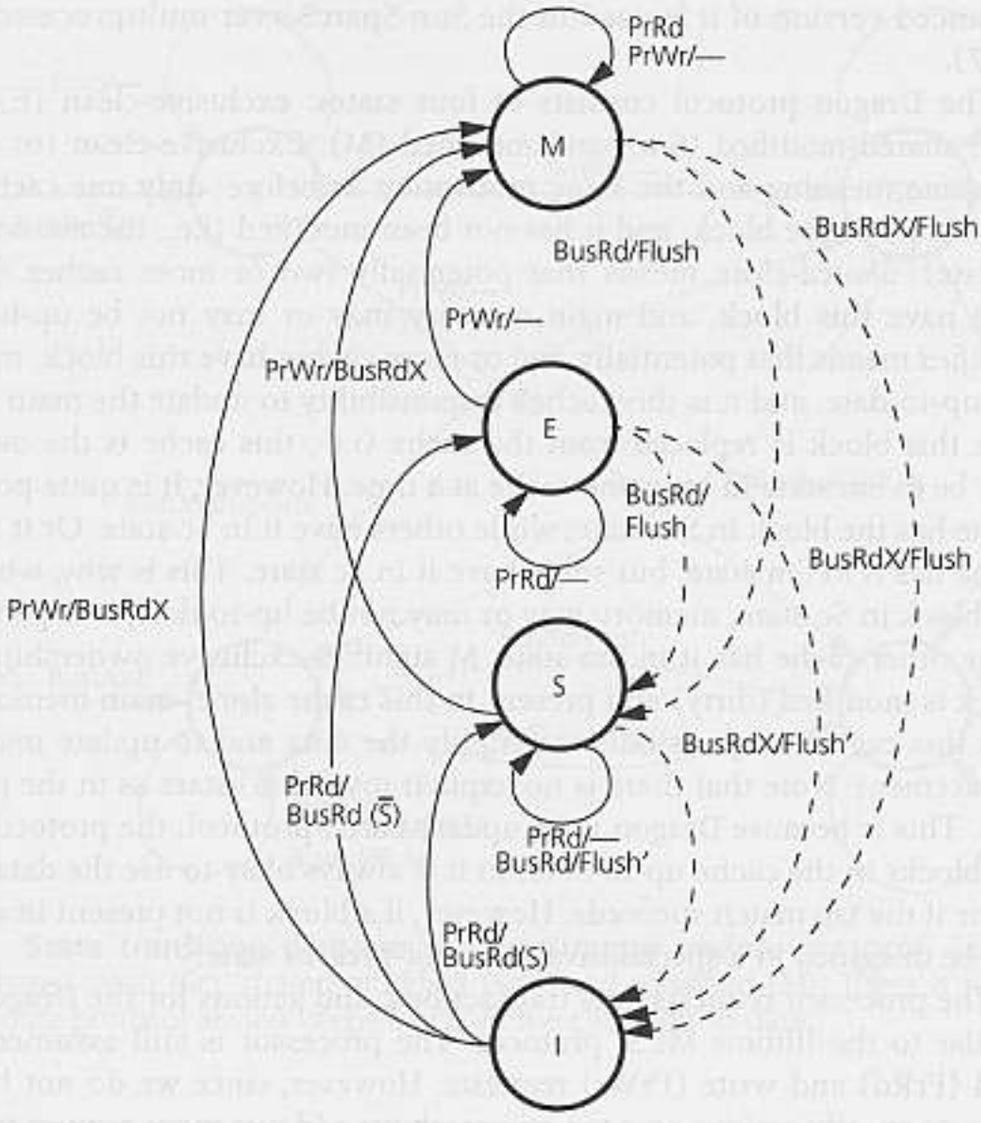

FIGURE 5.15 State transition diagram for the Illinois MESI protocol. MESI stands for the modified (dirty), exclusive, shared, and invalid states, respectively. The notation is the same as that in Figure 5.13. The E state helps reduce bus traffic for sequential programs where data is not shared. Whenever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

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

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

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

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

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

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

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

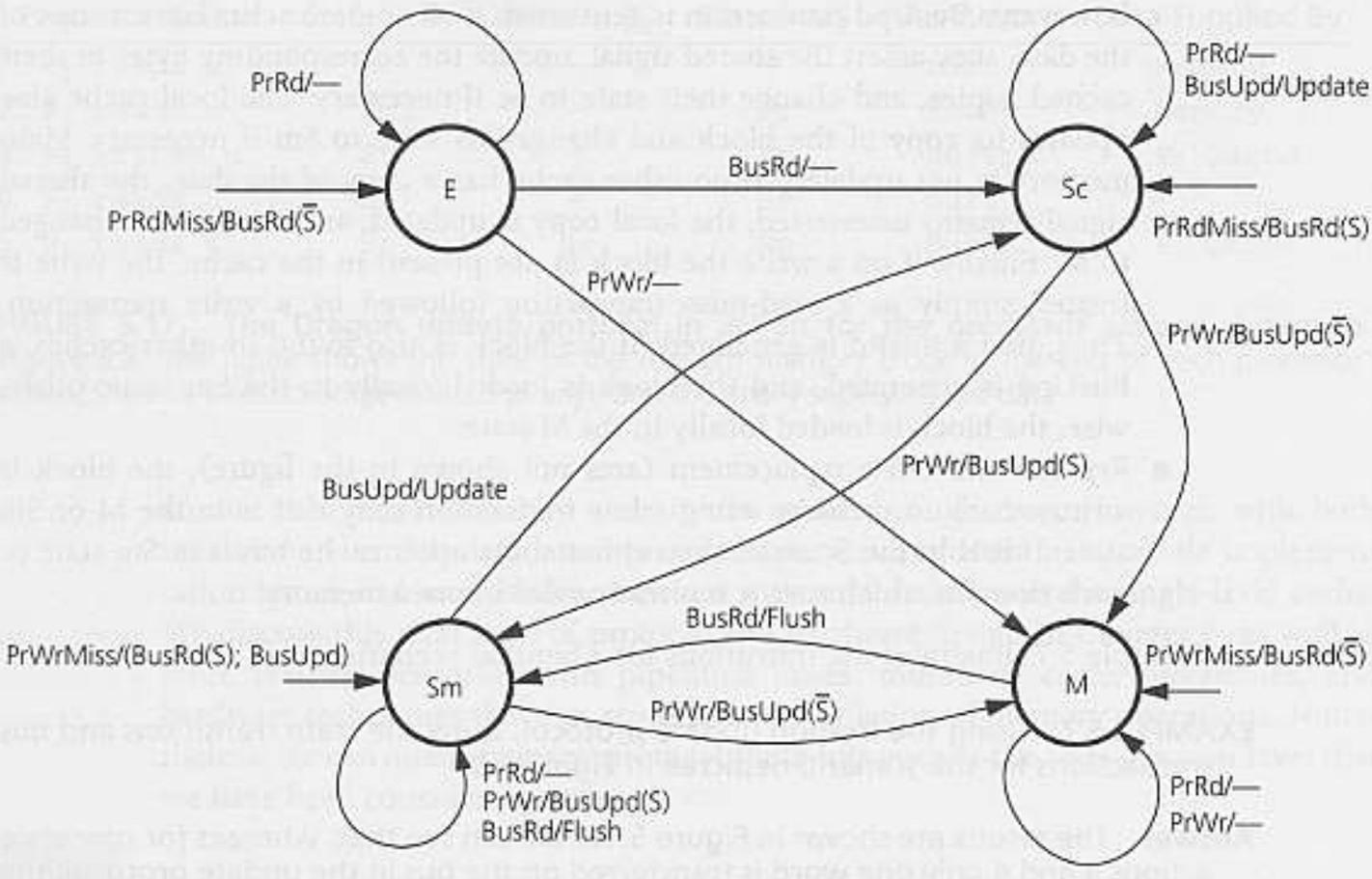

FIGURE 5.16 State transition diagram for the Dragon update protocol. The four states are exclusive (E), shared-clean (Sc), shared-modified (Sm), and modified (M). There is no invalid (I) state because the update protocol always keeps blocks in the cache up-to-date.

State Transitions

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

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

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

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

Example 5.7 illustrates the transitions for a familiar scenario.

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

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

Lower-Level Design Choices

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

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

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

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

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

5.4 ASSESSING PROTOCOL DESIGN TRADE-OFFS

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

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