I

Implementing Coherence Protocols

The devil is in the details.

Classic Proverb

I.1 Implementation Issues for the Snooping Coherence Protocol

The major complication in actually using the snooping coherence protocol from Section 6.3 is that write misses are not atomic: The operation of detecting a write miss, obtaining the bus, getting the most recent value, and updating the cache cannot be done as if it took a single cycle. In particular, two processors cannot both use the bus at the same time. Thus, we must decompose the write into several steps that may be separated in time, but will still preserve correct execution. The first step detects the miss and requests the bus. The second step acquires the bus, places the miss on the bus, gets the data, and completes the write. Each of these two steps is atomic, but the cache block does not become exclusive until the second step has begun. As long as we do not change the block to exclusive or allow the cache update to proceed before the bus is acquired, writes to the same cache block will serialize when they reach the second step of the coherence protocol. Unfortunately, this two-step process does introduce new complications in the protocol.

Figure I.1 shows the actual finite-state diagram for implementing coherence for this two-step process under the assumption that a bus transaction is atomic once the bus is acquired. This assumption simply means that the bus is not a split transaction, and once it is acquired any requests are processed before another processor can acquire the bus. We discuss the complexities of a split-transaction bus shortly. In the simplest implementation, the finite-state machine in Figure I.1 is simply replicated for each block in the cache. Since there is no interaction among operations on different cache blocks, this replication of the controller works. Replicating the controller is not necessary, but before we see why, let's make sure we understand how the finite-state controller in Figure I.1 operates.

The additional states in Figure I.1 over those in Figure 6.12 on page 559 are all transient: The controller will leave those states when the bus is available. Four of the states are pending write-back states that arise because in a write-back cache when a block is replaced (or invalidated) it must be written back to the memory. Four events can cause such a write back:

1. A write miss on the bus by another processor for this exclusive block.
2. A CPU read miss that forces the exclusive block to be replaced.
3. A CPU write miss that forces the exclusive block to be replaced.
4. A read miss on the bus by another processor for this block.

In each of the cases the next state differs, hence there are four separate pending write-back states with four different successor states.

Logically replicating the controller for each cache block allows correct operation if two conditions hold (in addition to our base assumption that the processor blocks until a cache access completes):

1. An operation on the bus for a cache block and a pending operation for a different cache block are noninterfering.

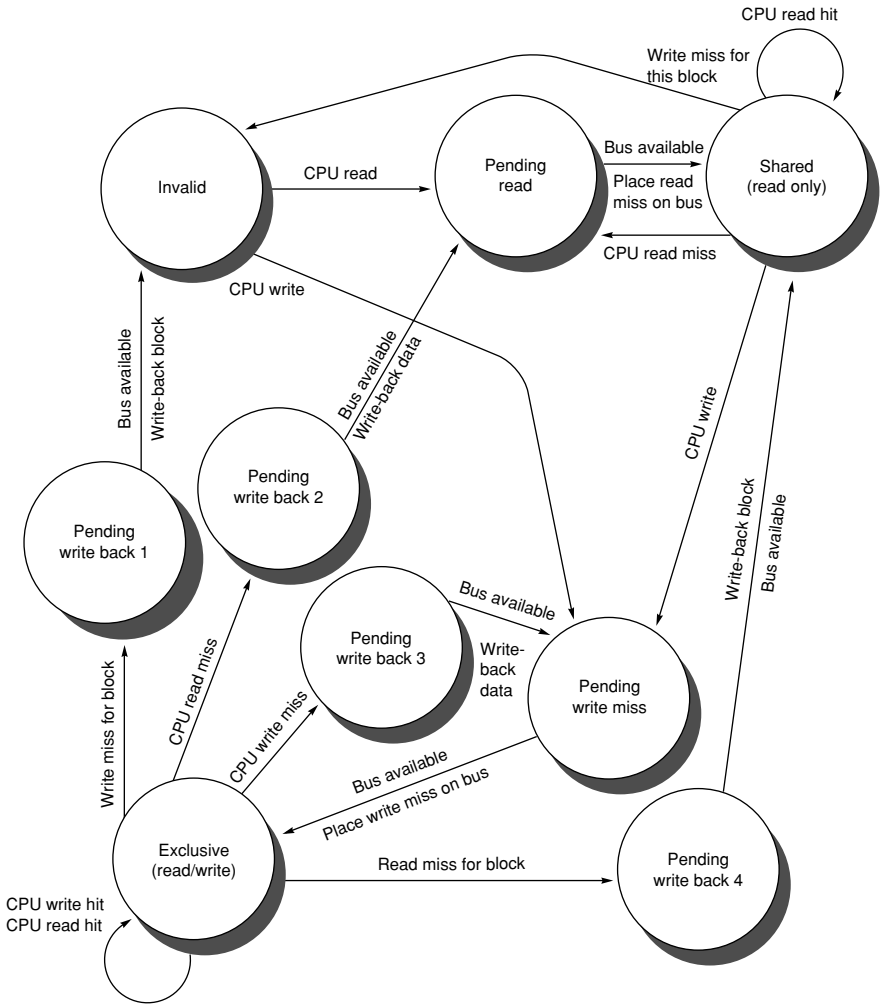

Figure I.1 A finite-state controller for a simple cache coherence scheme with a write-back cache. The engine that implements this controller must be reentrant, that is, it must handle multiple requests for different cache lines that are overlapped in time. The diagram assumes the processor stalls until a request is completed, but other transactions must be handled. This controller also assumes that a transition to a new state that involves a bus access does not complete until the bus access is completed. Notice that if we did not require a processor to generate a write miss when it transitioned from the shared to exclusive state, it might not obtain the latest value of a cache block, since some other processor may have updated that block. In a protocol using ownership or upgrade transitions, we will need to be able to transition out of the pending write state and restart an access if a conflicting write obtains the bus first.

2. The controller in Figure I.1 correctly deals with the cases when a pending operation and a bus operation are for the same block.

The first condition is certainly true, since operations for different blocks may proceed in any order and do not affect the state transitions for the other block. To see why the second condition is true, consider each of the pending states and what happens if a conflicting access occurs:

- Pending write back 1—The cache is writing back the data to eliminate it anyway, so a read or write miss for the block has no new effect. Notice, however, that the pending cache *must* use the bus cycle generated by the read or write miss to complete the write back. Otherwise, there will be no response to the miss, since the pending cache still has the only copy of the cache block. When it sees that the address of a miss matches the address of the block it is waiting to write back, it recognizes that the bus is available, writes the data, and transitions its state. This applies to all the pending write-back states.
- Pending write back 2, 3—The cache is eliminating a block in the exclusive state, so another miss for that block simply allows the write back to occur immediately. If the read or write miss on the bus is for the new block that the processor is trying to share, there is no interaction, since the processor does not yet have a copy of the block.
- Pending write back 4—In this case the processor is surrendering an exclusive block and simply completes the write back.
- Pending read, pending write miss —The processor does not yet have a copy of the block that it is waiting for, so a read or write miss for that block has no effect. Since the waiting cache still needs to place a miss on the bus and fetch the block, it is guaranteed to get a new copy.

With these additional states and our assumptions that the bus operates atomically, that misses always cause the state to be updated, and that the processor blocks until an access completes, our coherence implementation is both deadlock-free and correct. If some fairness guarantee is made for bus access, then this controller is also free of *livelock*. Livelock occurs when some portion of a computation cannot make progress, though other portions can. If one processor could be denied the bus indefinitely, then that processor could never make progress in its computation. Some guarantee of fairness on bus access prevents this.

There is still, however, one more critical implementation detail related to the bus transactions and what happens when a miss is processed. The key difference between the cache coherence case and the standard uniprocessor case occurs when the block is exclusive in some cache. Because it is a write-back cache, the memory copy is stale. In this case, the coherence unit will retrieve the block (called an *intervention*) and generate a write back. Since the memory does not know the state of the block, it will attempt to respond to the request as well. Since the data have been updated, the cache and processor will each attempt to drive the bus with different values. To prevent this, a line is added to the bus (often called the shared line) to coordinate the response. When the processor detects that it has a copy in the exclusive state, it signals the memory on this line and the memory

aborts the transaction. When the write back occurs, the memory gets the data and updates its copy. Since it is difficult to bound the amount of time that it can take to snoop the local cache copy, this line is usually implemented as a wired-OR with each processor holding its input low until it knows it does not have the block in exclusive state. The memory waits for the line to go high, indicating that no cache has the copy in the exclusive state, before putting data on the bus.

If the bus had a split-transaction capability then we could not assume that a response would occur immediately. In fact, implementing a split transaction with coherence is significantly more complex. One complication arises from the fact that we must number and track bus transactions, so that a controller knows when a bus action is a response to its request. Another complication is dealing with races that can arise because two operations for the same cache block could potentially be outstanding simultaneously. An example illustrates this complication best. What happens when two processors try to write a word in the same cache block? Without split transactions, one of the operations reaches the bus first and the other must change the state of the block to invalid and try the operation again. Only one of the transactions is outstanding on the bus at any point.

Example Suppose we have a split-transaction bus and no cache has a copy of a particular block. Show how when both P1 and P2 try to write a word in that block, we can get an incorrect result using the protocol in Figure I.1 on page I-3.

Answer With the protocol in Figure I.1, the following sequence of events could occur:

1. P1 places a write miss for the block on the bus. Since P2 has the data in the invalid state, nothing occurs.
2. P2 places its write miss on the bus; again, since no copy exists, no state changes are needed.
3. The memory responds to P1's request. P1 places the block in the exclusive state and writes the word into the block.
4. The memory responds to P2's request. P2 places the block in the exclusive state and writes the word into the block.

Disaster! Two caches now have the same block in the exclusive state and memory will be inconsistent.

How can this race be avoided? The simplest way is to use the broadcast capability of the bus. All coherence controllers track all bus accesses. In a split-transaction bus, the transactions must be tagged with the processor identity (or a transaction number), so that a processor can identify a reply to its request. Every controller can keep track of the memory address of any outstanding bus requests, since it can see the request and the corresponding reply on the bus. When the local processor generates a miss, the controller does not place the miss request on the bus until there are no outstanding requests for the same cache block. This will

force P2 in the above example to wait for P1's access to complete, allowing P1 to place the data in the exclusive state (and write the word into the block). The miss request from P2 will then cause P1 to do a write back and move the block to the invalid state. Alternatively, we could have each processor buffer only its own requests and track the responses to others. If the address of the requested block were included in the reply, then the second processor to request the block could ignore the reply and reissue its request.

These race conditions are what make implementing coherence even more tricky as the interconnection mechanism becomes more sophisticated. As we will see in the next section, such problems are slightly worse in a directory-based system that does not have a broadcast mechanism like a bus, which can be used to order all accesses.

I.2 Implementation Issues in the Distributed Directory Protocol

One further source of complexity of a directory protocol comes from the lack of atomicity in transactions. Several of the operations that are atomic in a bus-based snoopy protocol cannot be atomic in a directory-based machine. For example, a read miss, which is atomic in the snoopy protocol, cannot be atomic, since it requires messages to be sent to remote directories and caches. In fact, if we attempt to implement these operations in an atomic fashion in a distributed-memory machine, we can have deadlock. Recall from Chapter 6 that a deadlock means that the machine has reached a state from which it cannot make forward progress. This is easy to see with an example.

Example Show how deadlock can occur if a node treats a read miss as atomic and hence is unable to respond to other requests until the read miss is completed.

Answer Assume that two nodes P1 and P2 each have exclusive copies of cache blocks X1 and X2 that have different home directories. Consider the following sequence of events shown in Figure I.2.

Events caused by P1 activity	Events caused by P2 activity
P1 read miss for X2	P2 read miss for X1
Directory for X2 receives read miss and generates a fetch that is sent to P2	Directory for X1 receives read miss and generates a fetch that is sent to P1
Fetch arrives at P1, waits for completion of atomic read miss	Fetch arrives at P2, waits for completion of atomic read miss

Figure I.2 Events caused by P1 and P2 leading to deadlock.

At this point the nodes are deadlocked. In this case, since the requests are for separate blocks, deadlock can be avoided by duplicating the controller for each block. This allows the controllers to accept a request for one block while a request for another block is in process. In practice, complications arise because requests for the same block can collide, as we will see shortly.

The almost complete lack of atomicity in transactions causes most of the complexities in translating these state transition diagrams into actual finite-state controllers. There are two assumptions about the interconnection network that significantly simplify the implementation. First, we assume that the network provides point-to-point *in-order delivery* of messages. This means that two messages sent from a single node to another node arrive in the order they were sent. No assumptions are made about messages originating from, or destined to, different nodes. Second, we assume the network has unlimited buffering. This second assumption means that a message can always be accepted into the network. This reduces the possibility for deadlock and allows us to treat some nonatomic action, where we would need to be able to deal with a full set of network buffers, as atomic actions. Of course, we also assume that the network delivers all messages within a finite time.

While the first assumption, in-order transmission, is quite reasonable and is, in fact, true in many machines, the second assumption, unlimited buffering, is not true. Actually, the network need only be capable of buffering a finite number of messages, since we still assume that processors block on misses. In practice, this number may still be large and unreasonable, so later in the section we will discuss what has to change to eliminate the assumption that a message can always be accepted, while still preventing deadlock.

We also assume that the coherence controller is duplicated for each cache block (to avoid having to deal with unrelated transactions) and that a state transition only completes when a message has been transmitted and a data value reply received (when needed). This last assumption simply means that we do not allow the CPU to continue and read or write a cache block until the read or write miss is satisfied by a data value reply message. This simply eliminates a transition state that waits for the block to arrive. Because we are assuming unlimited buffering, we also assume that an outgoing message can always be transmitted before the next incoming message is accepted.

Under these assumptions the state transition diagram of Figure 6.29 on page 581 can be used for the coherence controller at the cache with one small addition: The controller simply throws away any incoming transactions, other than the data value reply, while waiting for a read or write miss. Let's look at each possible case that can arise while the cache is waiting for a response from the directory. Cases where the cache is transitioning the block to invalid, either from the shared or exclusive state, do not matter, since any incoming signals for this block do not affect the block once it is invalid. Hence, we need only consider cases where the processor is transitioning to the shared or exclusive state. There are two such cases:

- CPU read miss from either invalid or exclusive—The directory will not reply until the block is available. Furthermore, since any write back of an exclusive entry for this block has been done, the controller can ignore any requests.
- CPU write miss—Any required write back is done first and the processor is stalled. Since it cannot hold a block exclusive in this cache entry, it can ignore requests for this block until the write miss is satisfied from the directory.

The directory case is more complex to explain, since multiple cache controllers may send a message for the same block close to the same time. These operations must be serialized. Unlike the snoopy case where every controller sees every request on the bus at the same time, the individual caches only know what has happened when they are notified by the directory. Because the directory serializes the messages when it receives them and because all write misses for a given cache block go to the same directory, writes will be serialized by the home directory.

Thus, the directory controller's main problem is to deal with the distributed representation of the cache state. Since the directory must wait for the completion of certain operations, such as sending invalidates and fetching a cache block before transitioning state, most potential races are eliminated. Because we assume unlimited buffering, the directory can always complete a transaction before accepting the next incoming message. For this reason, the state transition diagram in Figure 6.30 can be used as an implementation. To see why, we must consider cases where the directory and the local cache do not agree on the state of a block. The cache can only have a block in a less restricted state than the directory believes the block is in, because transitioning to exclusive from invalid or shared, or to shared from invalid, requires a message to the directory and a reply. Thus, the only cases to consider are

- Local cache state is invalid, directory state is exclusive—The cache controller must have performed a data write back of the block (see Figure 6.29). Hence the directory will shortly obtain the block. Furthermore no invalidation is needed, since the block has been replaced.
- Local cache state is invalid, directory state is shared (the local cache is replacing the line)—The directory will send an invalidate, which may be ignored, since the block has been replaced. Some directory protocols send a *replacement hint* message when a shared line is replaced. Such messages are used to eliminate unnecessary invalidates and to reduce the state needed in the directory.
- Local cache state is shared, directory state is exclusive—The write back has already been done and the block has been replaced, so a fetch/invalidate, which could be sent by the directory, can be ignored.

Hence, the protocol operates correctly with infinite buffering.

Dealing with Finite Buffering

What happens when the network does not have unlimited buffering? The major implication of this limit is that a cache or directory controller may be unable to complete a message send. This could lead to deadlock. The example on page I-6 showed such a deadlock case. Even if we assume a separate controller for each cache block, so that the requests do not interfere in the controller, the example will deadlock if there are no buffers available to send the replies.

The occurrence of such a deadlock is based on three properties, which characterize many deadlock situations:

1. More than one resource is needed to complete a transaction: Buffers are needed to generate requests, create replies, and accept replies.
2. Resources are held until a nonatomic transaction completes: The buffer used to create the reply cannot be freed until the reply is accepted.
3. There is no global partial order on the acquisition of resources: Nodes can generate requests and replies at will.

These characteristics lead to deadlock, and avoiding deadlock requires breaking one of these properties. Imposing a global partial order, the solution used in a bus-based system, is unworkable in a larger-scale, distributed machine. Freeing up resources without completing a transaction is difficult, since the transaction must be completely backed out and cannot be left half-finished. Hence, our approach will be to try to resolve the need for multiple resources. We cannot simply eliminate this need, but we can try to ensure that the resources will always be available.

One way to ensure that a transaction can always complete is to guarantee that there are always buffers to accept messages. Although this is possible for a small machine with processors that block on a cache miss, it may not be very practical, since a single write could generate many invalidate messages. In addition, features such as prefetch would increase the amount of buffering required. There is an alternative strategy, which most systems use, and which ensures that a transaction will not actually be initiated until we can guarantee that it has the resources to complete. The strategy has four parts:

1. A separate network (physical or virtual) is used for requests and replies, where a reply is any message that a controller waits for in transitioning between states. This ensures that new requests cannot block replies that will free up buffers.
2. Every request that expects a reply allocates space to accept the reply when the request is generated. If no space is available, the request waits. This ensures that a node can always accept a reply message, which will allow the replying node to free its buffer.

3. Any controller can reject (usually with a *negative acknowledge* or *NAK*) any request, but it can never NAK a reply. This prevents a transaction from starting if the controller cannot guarantee that it has buffer space for the reply.
4. Any request that receives a NAK in response is simply retried.

To understand why this is sufficient to prevent deadlock, let's first consider our earlier example. Because a write miss is a request that requires a reply, the space to accept the reply is preallocated. Hence, both nodes will have space for the reply. Since the networks are separate, a reply can be received even if no more space is available for requests. Since the requests are for two different blocks, the separate coherence controllers handle the requests. If the accesses are for the same address, then they are serialized at the directory and no problem exists.

To see that there are no deadlocks more generally, we must ensure that all replies can be accepted, and that every request is eventually serviced. Since a cache controller or directory controller can have at most one request needing a reply outstanding, it can always accept the reply when it returns. To see that every request is eventually serviced, we need only show that any request could be completed. Since every request starts with a read or write miss at a cache, it is sufficient to show that any read or write miss is eventually serviced. Since the write miss case includes the actions for a read miss as a subset, we focus on showing the write misses are serviced. The simplest situation is when the block is uncached; since that case is subsumed by the case when the block is shared, we focus on the shared and exclusive cases. Let's consider the case where the block is shared:

- The CPU attempts to do a write and generates a write miss that is sent to the directory. At this point the processor is stalled.
- The write miss is sent to the directory controller for this memory block. Note that although one cache controller handles all the requests for a given cache block, regardless of its memory contents, there is a controller for every memory block. Thus the only conflict at the directory controller is when two requests arrive for the same block. This is critical to the deadlock-free operation of the controller and needs to be addressed in an implementation using a single controller.
- Now consider what happens at the directory controller: Suppose the write miss is the next thing to arrive at the directory controller. The controller sends out the invalidates, which can always be accepted if the controller for this block is idle. If the controller is not idle, then the processor must be stalled. Since the processor is stalled, it must have generated a read or write miss. If it generated a read miss, then it has either displaced this block or does not have a copy. If it does not have a copy, then it has sent a read miss and cannot continue until the read miss is processed by the directory (the read miss will not be handled until the write miss is). If the controller has replaced the block, then we need not worry about it. If the controller is idle, then an invalidate occurs, and the copy is eliminated.

The case where the block is exclusive is somewhat trickier. Our analysis begins when the write miss arrives at the directory controller for processing. There are two cases to consider:

- The directory controller sends a fetch/invalidate message to the processor where it arrives to find the cache controller idle and the block in the exclusive state. The cache controller sends a data write back to the home directory and makes its state invalid. This reply arrives at the home directory controller, which can always accept the reply, since it preallocated the buffer. The directory controller sends back the data to the requesting processor, which can always accept the reply; after the cache is updated the requesting cache controller restarts the processor.
- The directory controller sends a fetch/invalidate message to the node indicated as owner. When the message arrives at the owner node, it finds that this cache controller has taken a read or write miss that caused the block to be replaced. In this case, the cache controller has already sent the block to the home directory with a data write back and made the data unavailable. Since this is exactly the effect of the fetch/invalidate message, the protocol operates correctly in this case as well.

We have shown that our coherence mechanism operates correctly when controllers are replicated and when responses can be NAKed and retried. Both of these assumptions generate some problems in the implementation.

Implementing the Directory Controllers

First, let's consider how these controllers, which we have assumed are replicated, can be built without actually replicating them. On the side of the cache controllers, because the processors stall, the actual implementation is quite similar to what was needed for the snoopy controller. We can simply add the transient states just as we did for the snoopy case and note that a transaction for a different cache block can be handled while the current processor-generated operation is pending. Since a processor blocks on a request, at most one pending operation need be dealt with.

On the side of the directory controller, things are more complicated. The difficulty arises from the way we handle the retrieval and return of a block. In particular, during the time a directory retrieves an exclusive block and returns it to the requesting node, the directory must accommodate other transactions. Otherwise, integrating the directory controllers for different cache blocks will lead to the possibility of deadlock. Because of this situation, the directory controller must be reentrant, that is, it must be capable of suspending its execution while waiting for a reply and accept another transaction. The only place this must occur is in response to read or write misses, while waiting for a response from the owner. This leads to three important observations:

1. The state of the controller need only be saved and restored while either a fetch or a fetch/invalidate operation is outstanding.
2. The implementation can bound the number of outstanding transactions being handled in the directory, by simply NAKing read or write miss requests that could cause the number of outstanding requests to be exceeded.
3. If instead of returning the data through the directory, the owner node forwards the data directly to the requester (as well as returning it to the directory), we can eliminate the need for the directory to handle more than one outstanding request. This motivation, in addition to the reduction of latency, is the reason for using the forwarding style of protocol. The forwarding-style protocol introduces another type of problem that we discuss in the exercises.

The major remaining implementation difficulty is to handle NAKs. One alternative is for each processor to keep track of its outstanding transactions, so it knows, when the NAK is received, what the requested transaction was. The alternative is to bundle the original request into the NAK, so that the controller receiving the NAK can determine what the original request was. Because every request allocates a slot to receive a reply and a NAK is a reply, NAKs can always be received. In fact, the buffer holding the return slot for the request can also hold information about the request, allowing the processor to reissue the request if it is NAKed.

This completes the implementation of the directory scheme. In practice, great care is required to implement these protocols correctly and to avoid deadlock. The key ideas we have seen in this section—dealing with nonatomicity and finite buffering—are critical to ensuring a correct implementation. Designers have found that both formal and informal verification techniques are helpful for ensuring that implementations are correct.

Exercises

I.1 [20] <6.5, I.2> The Convex Exemplar is a coherent shared-memory machine organized as a ring of eight-processor clusters. Describe a protocol for this machine, assuming that the ring can be snooped and that a directory sits at the junction of the ring and can also be interrogated from inside the cluster. How much directory storage is needed? If the coherence misses are uniformly distributed and the capacity misses are all within a cluster, what is the average memory access time for Ocean running on 64 processors?

I.2 [15/20] <6.5, I.2> As we discussed in Section I.2, many DSM machines use a forwarding protocol, where a write miss request to a remote dirty block is forwarded to the node that has the copy of the block. The remote node then generates both a write-back operation and a data value reply.

a. [15] <6.5> Modify the state diagrams of Figures 6.29 and 6.30 so that the diagrams implement a forwarding protocol.

b. [20] <6.5, I.2> Forwarding protocols introduce a race condition into the protocol. Describe this race condition. Show how NAKs can be used to resolve the race condition.

I.3 [20] <6.5, I.2> Supporting lock-up free caches can have different implications for coherence protocols. Show how, without additional changes, allowing multiple outstanding misses from a node in a DSM can lead to deadlock—even if buffering is unlimited.