CHAPTER 6

Coherence Protocols

In this chapter, we return to the topic of cache coherence that we introduced in Chapter 2. We defined coherence in Chapter 2, in order to understand coherence's role in supporting consistency, but we did not delve into how specific coherence protocols work or how they are implemented. This chapter discusses coherence protocols in general, before we move on to specific classes of protocols in the next two chapters. We start in Section 6.1 by presenting the big picture of how coherence protocols work, and then show how to specify protocols in Section 6.2. We present one simple, concrete example of a coherence protocol in Section 6.3 and explore the protocol design space in Section 6.4.

6.1 THE BIG PICTURE

The goal of a coherence protocol is to maintain coherence by enforcing the invariants introduced in Section 2.3 and restated here:

1. *Single-Writer, Multiple-Read (SWMR) Invariant*. For any memory location A, at any given (logical) time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.
2. *Data-Value Invariant*. The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

To implement these invariants, we associate with each storage structure—each cache and the LLC/memory—a finite state machine called a *coherence controller*. The collection of these coherence controllers constitutes a distributed system in which the controllers exchange messages with each other to ensure that, for each block, the SWMR and data value invariants are maintained at all times. The interactions between these finite state machines are specified by the coherence protocol.

Coherence controllers have several responsibilities. The coherence controller at a cache, which we refer to as a *cache controller*, is illustrated in Figure 6.1. The cache controller must service requests from two sources. On the "core side," the cache controller interfaces to the processor core.

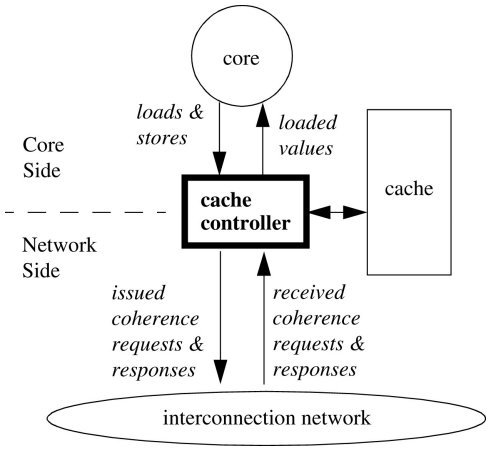

FIGURE 6.1: Cache controller.

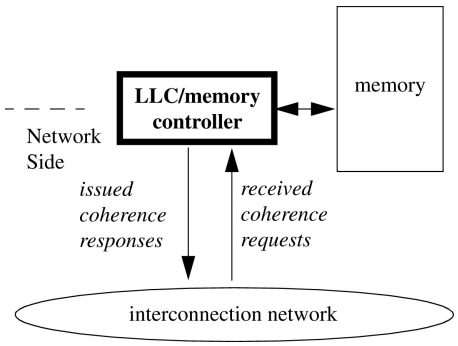

FIGURE 6.2: Memory controller.

The controller accepts loads and stores from the core and returns load values to the core. A cache miss causes the controller to initiate a coherence *transaction* by issuing a coherence *request* (e.g., request for read-only permission) for the block containing the location accessed by the core. This coherence request is sent across the interconnection network to one or more coherence controllers. A transaction consists of a request and the other message(s) that are exchanged in order to satisfy the request (e.g., a data response message sent from another coherence controller to the requestor). The types of transactions and the messages that are sent as part of each transaction depend on the specific coherence protocol.

On the cache controller's "network side," the cache controller interfaces to the rest of the system via the interconnection network. The controller receives coherence requests and coherence responses that it must process. As with the core side, the processing of incoming coherence messages depends on the specific coherence protocol.

The coherence controller at the LLC/memory, which we refer to as a *memory controller*, is illustrated in Figure 6.2. A memory controller is similar to a cache controller, except that it usually has only a network side. As such, it does not issue coherence requests (on behalf of loads or stores) or receive coherence responses. Other agents, such as I/O devices, may behave like cache controllers, memory controllers, or both depending upon their specific requirements.

Each coherence controller implements a set of finite state machines—logically one independent, but identical finite state machine per block—and receives and processes *events* (e.g., incoming coherence messages) depending upon the block's state. For an event of type E (e.g., a store request

from the core to the cache controller) to block B, the coherence controller takes actions (e.g., issues a coherence request for read-write permission) that are a function of E and of B's *state* (e.g., read-only). After taking these actions, the controller may change the state of B.

6.2 SPECIFYING COHERENCE PROTOCOLS

We specify a coherence protocol by specifying the coherence controllers. We could specify coherence controllers in any number of ways, but the particular behavior of a coherence controller lends itself to a tabular specification [9]. As shown in Table 6.1, we can specify a controller as a table in which rows correspond to block states and columns correspond to events. We refer to a state/event entry in the table as a *transition*, and a transition for event E pertaining to block B consists of (a) the actions taken when E occurs and (b) the next state of block B. We express transitions in the format "action/next state" and we may omit the "next state" portion if the next state is the current state. As an example of a transition in Table 6.1, if a store request for block B is received from the core and block B is in a read-only state (RO), then the table shows that the controller's transition will be to perform the action "issue coherence request for read-write permission [to block B]" and change the state of block B to RW.

The example in Table 6.1 is intentionally incomplete, for simplicity, but it illustrates the capability of a tabular specification methodology to capture the behavior of a coherence controller. To specify a coherence protocol, we simply need to completely specify the tables for the cache controllers and the memory controllers.

TABLE 6.1: Tabular Specification Methodology. This is an Incomplete Specification of a Cache Coherence Controller. Each Entry in the Table Specifies the Actions Taken and the Next State of the Block.

States	Events: load request from core	Events: store request from core	Events: incoming coherence request to obtain block in read-write state
not readable or writeable (N)	issue coherence request for read-only permission/RO	issue coherence request for read-write permission/RW	<no action>
read-only (RO)	give data from cache to core	issue coherence request for read-write permission/RW	<no action>/N
read-write (RW)	give data from cache to core/RO	write data to cache	send block to requestor/N

The differences between coherence protocols lie in the differences in the controller specifications. These differences include different sets of block states, transactions, events, and transitions. In Section 6.4, we describe the coherence protocol design space by exploring the options for each of these aspects, but we first specify one simple, concrete protocol.

6.3 EXAMPLE OF A SIMPLE COHERENCE PROTOCOL

To help understand coherence protocols, we now present a simple protocol. Our system model is the baseline system model from Section 2.1, but with the interconnection network restricted to being a shared bus: a shared set of wires on which a core can issue a message and have it observed by all cores and the LLC/memory.

Each cache block can be in one of two stable coherence states: I(nvalid) and V(alid). Each block at the LLC/memory can also be in one of two coherence states: I and V. At the LLC/memory, the state I denotes that all caches hold the block in state I, and the state V denotes that one cache holds the block in state V. There is also a single transient state for cache blocks, IV^D, discussed below. At system startup, all cache blocks and LLC/memory blocks are in state I. Each core can issue load and store requests to its cache controller; the cache controller will implicitly generate an Evict Block event when it needs to make room for another block. Loads and stores that miss in the cache initiate coherence transactions, as described below, to obtain a valid copy of the cache block. Like all the protocols in this primer, we assume a writeback cache; that is, when a store hits, it writes the store value only to the (local) cache and waits to write the entire block back to the LLC/memory in response to an Evict Block event.

There are two types of coherence transactions implemented using three types of bus messages: *Get* requests a block, *DataResp* transfers the block's data, and *Put* writes the block back to the memory controller. On a load or store miss, the cache controller initiates a Get transaction by sending a Get message and waiting for the corresponding DataResp message. The Get transaction is atomic in that no other transaction (either Get or Put) may use the bus between when the cache sends the Get and when the DataResp for that Get appears on the bus. On an Evict Block event, the cache controller sends a Put message, with the entire cache block, to the memory controller.

We illustrate the transitions between the stable coherence states in Figure 6.3. We use the prefaces "Own" and "Other" to distinguish messages for transactions initiated by the given cache controller versus those initiated by other cache controllers. Note that if the given cache controller has the block in state V and another cache requests it with a Get message (denoted Other-Get), the owning cache must respond with a block (using a DataResp message, not shown) and transition to state I.

Table 6.2 and Table 6.3 specify the protocol in more detail. Shaded entries in the table denote impossible transitions. For example, a cache controller should never see its own Put request on the bus for a block that is in state V in its cache (as it should have already transitioned to state I).

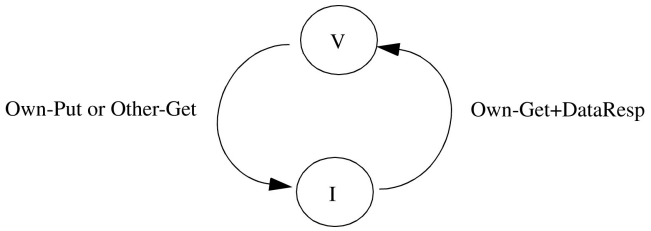

FIGURE 6.3: Transitions between stable states of blocks at cache controller.

The transient state IV^D corresponds to a block in state I that is waiting for data (via a DataResp message) before transitioning to state V. Transient states arise when transitions between stable states are not atomic. In this simple protocol, individual message sends and receives are atomic, but fetching a block from the memory controller requires sending a Get message and receiving a DataResp message, with an indeterminate gap in between. The IV^D state indicates that the protocol is waiting for a DataResp. We discuss transient states in more depth in Section 6.4.1.

This coherence protocol is simplistic and inefficient in many ways, but the goal in presenting this protocol is to gain an understanding of how protocols are specified. We use this specification methodology throughout this book when presenting different types of coherence protocols.

TABLE 6.2: Cache Controller Specification. Shaded Entries are Impossible and Blank Entries Denote Events That are Ignored.

States	Core Events		Bus Events					
			Messages for Own Transactions			Messages for Other Cores' Transactions		
	Load or Store	Evict Block	Own-Get	DataResp for Own-Get	Own-Put	Other-Get	DataResp for Other-Get	Other-Put
I	issue Get /IV^D							
IV^D	stall Load or Store	stall Evict		copy data into cache, perform Load or Store /V				
V	perform Load or Store	Issue Put (with data) /I				Send DataResp /I		

TABLE 6.3: Memory Controller Specification

State	Bus Events	
	Get	Put
I	send data block in DataResp message to requestor/V	
V		Update data block in memory/I

6.4 OVERVIEW OF COHERENCE PROTOCOL DESIGN SPACE

As mentioned in Section 6.1, a designer of a coherence protocol must choose the states, transactions, events, and transitions for each type of coherence controller in the system. The choice of stable states is largely independent of the rest of the coherence protocol. For example, there are two different classes of coherence protocols called snooping and directory, and an architect can design a snooping protocol or a directory protocol with the same set of stable states. We discuss stable states, independent of protocols, in Section 6.4.1. Similarly, the choice of transactions is also largely independent of the specific protocol, and we discuss transactions in Section 6.4.2. However, unlike the choices of stable states and transactions, the events, transitions and specific transient states are highly dependent on the coherence protocol and cannot be discussed in isolation. Thus, in Section 6.4.3, we discuss a few of the major design decisions in coherence protocols.

6.4.1 States

In a system with only one actor (e.g., a single core processor without coherent DMA), the state of a cache block is either valid or invalid. There might be two possible valid states for a cache block if there is a need to distinguish blocks that are *dirty*. A dirty block has a value that has been written more recently than other copies of this block. For example, in a two-level cache hierarchy with a write-back L1 cache, the block in the L1 may be dirty with respect to the stale copy in the L2 cache.

A system with multiple actors can also use just these two or three states, as in Section 6.3, but we often want to distinguish between different kinds of valid states. There are four characteristics of a cache block that we wish to encode in its state: validity, dirtiness, exclusivity, and ownership [10]. The latter two characteristics are unique to systems with multiple actors.

- Validity: A *valid* block has the most up-to-date value for this block. The block may be read, but it may only be written if it is also exclusive.

- Dirtiness: As in a single core processor, a cache block is *dirty* if its value is the most up-to-date value, this value differs from the value in the LLC/memory, and the cache controller is responsible for eventually updating the LLC/memory with this new value. The term *clean* is often used as the opposite of dirty.
- Exclusivity: A cache block is *exclusive*[1] if it is the only privately cached copy of that block in the system (i.e., the block is not cached anywhere else except perhaps in the shared LLC).
- Ownership: A cache controller (or memory controller) is the *owner* of a block if it is responsible for responding to coherence requests for that block. In most protocols, there is exactly one owner of a given block at all times. A block that is owned may not be evicted from a cache to make room for another block—due to a capacity or conflict miss—without giving the ownership of the block to another coherence controller. Non-owned blocks may be evicted silently (i.e., without sending any messages) in some protocols.

In this section, we first discuss some commonly used *stable states*—states of blocks that are not currently in the midst of a coherence transaction—and then discuss the use of *transient states* for describing blocks that are currently in the midst of transactions.

6.4.1.1 Stable States

Many coherence protocols use a subset of the classic five state MOESI model first introduced by Sweazey and Smith [10]. These MOESI (often pronounced either "MO-sey" or "mo-EE-see") states refer to the states of blocks in a cache, and the most fundamental three states are MSI; the O and E states may be used, but they are not as basic. Each of these states has a different combination of the characteristics described above.

- M(odified): The block is valid, exclusive, owned, and potentially dirty. The block may be read or written. The cache has the only valid copy of the block, the cache must respond to requests for the block, and the copy of the block at the LLC/memory is potentially stale.
- S(hared): The block is valid but not exclusive, not dirty, and not owned. The cache has a read-only copy of the block. Other caches may have valid, read-only copies of the block.
- I(nvalid): The block is invalid. The cache either does not contain the block or it contains a potentially stale copy that it may not read or write. In this primer, we do not distinguish between these two situations, although sometimes the former situation may be denoted as the "Not Present" state.

[1]The terminology here can be confusing, because there is a cache coherence state that is called "Exclusive," but there are other cache coherence states that are exclusive in the sense defined here.

The most basic protocols use only the MSI states, but there are reasons to add the O and E states to optimize certain situations. We will discuss these optimizations in later chapters when we discuss snooping and directory protocols with and without these states. For now, here is the complete list of MOESI states:

- M(odified)
- O(wned): The block is valid, owned, and potentially dirty, but not exclusive. The cache has a read-only copy of the block and must respond to requests for the block. Other caches may have a read-only copy of the block, but they are not owners. The copy of the block in the LLC/memory is potentially stale.
- E(xclusive): The block is valid, exclusive, and clean. The cache has a read-only copy of the block. No other caches have a valid copy of the block, and the copy of the block in the LLC/memory is up-to-date. In this primer, we consider the block to be owned when it is in the Exclusive state, although there are protocols in which the Exclusive state is not treated as an ownership state. When we present MESI snooping and directory protocols in later chapters, we discuss the issues involved with making Exclusive blocks owners or not.
- S(hared)
- I(nvalid)

We illustrate a Venn diagram of the MOESI states in Figure 6.4. The Venn diagram shows which states share which characteristics. All states besides I are valid. M, O, and E are ownership states. Both M and E denote exclusivity, in that no other caches have a valid copy of the block. Both

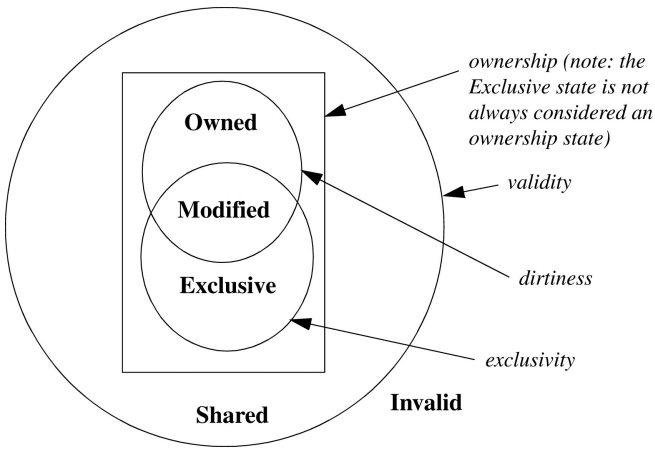

FIGURE 6.4: MOESI states.

M and O indicate that the block is potentially dirty. Returning to the simplistic example in Section 6.3, we observe that the protocol effectively condensed the MOES states into the V state.

The MOESI states, although quite common, are not an exhaustive set of stable states. For example, the F(orward) state is similar to the O state except that it is clean (i.e., the copy in the LLC/memory is up-to-date). There are many possible coherence states, but we focus our attention in this primer on the well-known MOESI states.

6.4.1.2 Transient States

Thus far we have discussed only the stable states that occur when there is no current coherence activity for the block, and it is only these stable states that are used when referring to a protocol (e.g., "a system with a MESI protocol"). However, as we saw even in the example in Section 6.3, there may exist transient states that occur during the transition from one stable state to another stable state. In Section 6.3, we had the transient state IV^D (in I, going to V, waiting for DataResp). In more sophisticated protocols, we are likely to encounter dozens of transient states. We encode these states using a notation XY^Z, which denotes that the block is transitioning from stable state X to stable state Y, and the transition will not complete until an event of type Z occurs. For example, in a protocol in a later chapter, we use IM^D to denote that a block was previously in I and will become M once a D(ata) message arrives for that block.

6.4.1.3 States of Blocks in the LLC/Memory

The states that we have discussed thus far—both stable and transient—pertain to blocks residing in caches. Blocks in the LLC and memory also have states associated with them, and there are two general approaches to naming states of blocks in the LLC and memory. The choice of naming convention does not affect functionality or performance; it is simply a specification issue that can confuse an architect unfamiliar with the convention.

- Cache-centric: In this approach, which we believe to be the most common, the state of a block in the LLC and memory is an aggregation of the states of this block in the caches. For example, if a block is in all caches in I, then the LLC/memory state for this block is I. If a block is in one or more caches in S, then the LLC/memory state is S. If a block is in a single cache in M, then the LLC/memory state is M.
- Memory-centric: In this approach, the state of a block in the LLC/memory corresponds to the *memory controller's* permissions to this block (rather than the permissions of the caches). For example, if a block is in all caches in I, then the LLC/memory state for this block is O (not I, as in the cache-centric approach), because the LLC/memory behaves like an owner

of the block. If a block is in one or more caches in S, then the LLC/memory state is also O, for the same reason. However, if the block is in a single cache in M or O, then the LLC/memory state is I, because the LLC/memory has an invalid copy of the block.

All protocols in this primer use *cache-centric* names for the states of blocks in the LLC and memory.

6.4.1.4 Maintaining Block State

The system implementation must maintain the states associated with blocks in caches, the LLC, and memory. For caches and the LLC, this generally requires extending the per-block cache state by at most a few bits, since the number of stable states is generally small (e.g., 5 states for a MOESI protocol requires 3 bits per block). Coherence protocols may have many more transient states, but need maintain these states only for those blocks that have pending coherence transactions. Implementations typically maintain these transient states by adding additional bits to the miss status handling registers (MSHRs) or similar structures that are used to track these pending transactions [4].

For memory, it might appear that the much greater aggregate capacity would pose a significant challenge. However, many current multicore systems maintain an inclusive LLC, which means that the LLC maintains a copy of every block that is cached anywhere in the system (even "exclusive" blocks). With an inclusive LLC, memory does not need to explicitly represent the coherence state. If a block resides in the LLC, its state in memory is the same as its state in the LLC. If the block is not in the LLC, its state in memory is implicitly Invalid, because absence from an inclusive LLC implies that the block is not in any cache. The sidebar discusses how memory state was maintained in the days before multicores with inclusive LLCs. The above discussion of memory assumes a system with a single multicore chip, as does most of this primer. Systems with multiple multicore chips may benefit from explicit coherence state logically at memory.

6.4.2 Transactions

Most protocols have a similar set of transactions, because the basic goals of the coherence controllers are similar. For example, virtually all protocols have a transaction for obtaining Shared (read-only) access to a block. In Table 6.4 we list a set of common transactions and, for each transaction, we describe the goal of the requestor that initiates the transaction. These transactions are all initiated by cache controllers that are responding to requests from their associated cores. In Table 6.5, we list the requests that a core can make to its cache controller and how these core requests can lead the cache controller into initiating coherence transactions.

Sidebar: Before Multicores: Maintaining Coherence State at Memory

Traditional, pre-multicore snooping protocols needed to maintain coherence state for each block of memory, and they could not use the LLC as explained in Section 6.4.1.4. We briefly discuss several ways of maintaining this state and the associated engineering tradeoffs.

Augment Each Block of Memory with State Bits. The most general implementation is to add extra bits to each block of memory to maintain the coherence state. If there are N possible states at memory, then each block needs $\log_2 N$ extra bits. Although, this design is fully general and conceptually straightforward, it has several drawbacks. First, the extra bits may increase cost in two ways. Adding two or three extra bits is difficult with modern block-oriented DRAM chips, which are typically at least 4-bits wide and frequently much wider. Plus any change in the memory precludes using commodity DRAM modules (e.g., DIMMs), which significantly increases cost. Fortunately, for protocols that require only a few bits of state per block it is possible to store these using a modified ECC code. By maintaining ECC on a larger granularity (e.g., 512 bits rather than 64 bits), it is possible to free up enough code space to "hide" a handful of extra bits while using commodity DRAM modules [5, 7, 1]. The second drawback is that storing the state bits in DRAM means that obtaining the state incurs the full DRAM latency, even in the case that the most recent version of the block is stored in some other cache. In some cases, this may increase the latency of cache-to-cache coherence transfers. Finally, storing the state in DRAM means that all state changes require a DRAM read-modify-write cycle, which may impact both power and DRAM bandwidth.

Add Single State Bit per Block at Memory. A design option used by the Synapse [3] was to distinguish the two stable states (I and V) using a single bit that is associated with every block of memory. Few blocks are ever in transient states, and those states can be maintained with a small dedicated structure. This design is a subset of the more complete first design, with minimal storage cost.

Zero-bit logical OR. To avoid having to modify memory, we can have the caches reconstruct the memory state on-demand. The memory state of a block is a function of the block's state in every cache, so, if all of the caches aggregate their state, they can determine the memory state. The system can infer whether the memory is the owner of a block by having all of the cores send an "IsOwned?[1]" signal to a logical OR gate (or tree of OR gates) with a number of inputs equal to the number of caches. If the output of this OR is high, it denotes that a cache is owner; if the output is low, then memory is the owner. This solution avoids the need for any state to be maintained in memory. However, implementing a fast OR, either with logic gates or a wired-OR, can be difficult.

[1]This IsOwned signal is not to be confused with the Owned cache state. The IsOwned signal is asserted by a cache in a state of ownership, which includes the Owned, Modified, and Exclusive cache states.

TABLE 6.4: Common Transactions.

Transaction	Goal of Requestor
GetShared (GetS)	obtain block in Shared (read-only) state
GetModified (GetM)	obtain block in Modified (read-write) state
Upgrade (Upg)	upgrade block state from read-only (Shared or Owned) to read-write (Modified); Upg (unlike GetM) does not require data to be sent to requestor
PutShared (PutS)	evict block in Shared state[a]
PutExclusive (PutE)	evict block in Exclusive state[a]
PutOwned (PutO)	evict block in Owned state
PutModified (PutM)	evict block in Modified state

a. Some protocols do not require a coherence transaction to evict a Shared block and/or an Exclusive block (i.e., the PutS and/or PutE are "silent").

Although most protocols use a similar set of transactions, they differ quite a bit in how the coherence controllers interact to perform the transactions. As we will see in the next section, in some protocols (e.g., snooping protocols) a cache controller initiates a GetS transaction by broadcasting a GetS request to all coherence controllers in the system, and whichever controller is currently the owner of the block responds to the requestor with a message that contains the desired data. Conversely, in other protocols (e.g., directory protocols) a cache controller initiates a GetS transaction by sending a unicast GetS message to a specific, pre-defined coherence controller that may either respond directly or may forward the request to another coherence controller that will respond to the requestor.

TABLE 6.5: Common Core Requests to Cache Controller.

Event	Response of (Typical) Cache Controller
load	if cache hit, respond with data from cache; else initiate GetS transaction
store	if cache hit in state E or M, write data into cache; else initiate GetM or Upg transaction
atomic read-modify-write	if cache hit in state E or M, atomically execute read-modify-write semantics; else initiate GetM or Upg transaction
instruction fetch	if cache hit (in I-cache), respond with instruction from cache; else initiate GetS transaction
read-only prefetch	if cache hit, ignore; else may optionally initiate GetS transaction[a]
read-write prefetch	if cache hit in state M, ignore; else may optionally initiate GetM or Upg transaction
replacement	depending on state of block, initiate PutS, PutE, PutO, or PutM transaction

a. A cache controller may choose to ignore a prefetch request from the core.

6.4.3 Major Protocol Design Options

There are many different ways to design a coherence protocol. Even for the same set of states and transactions, there are many different possible protocols. The design of the protocol determines what events and transitions are possible at each coherence controller; unlike with states and transactions, there is no way to present a list of possible events or transitions that is independent from the protocol.

Despite the enormous design space for coherence protocols, there are two primary design decisions that have a major impact on the rest of the protocol, and we discuss them next.

6.4.3.1 Snooping vs. Directory

There are two main classes of coherence protocols: snooping and directory. We present a brief overview of these protocols now and defer in-depth coverage of them until Chapter 7 and Chapter 8, respectively.

- Snooping protocol: A cache controller initiates a request for a block by broadcasting a request message to all other coherence controllers. The coherence controllers collectively "do the right thing," e.g., sending data in response to another core's request if they are the owner. Snooping protocols rely on the interconnection network to deliver the broadcast messages in a consistent order to all cores. Most snooping protocols assume that requests arrive in a total order, e.g., via a shared-wire bus, but more advanced interconnection networks and relaxed orders are possible.
- Directory protocol: A cache controller initiates a request for a block by unicasting it to the memory controller that is the *home* for that block. Each memory controller maintains a directory that holds state about each block in the LLC/memory, such as the identity of the current owner or the identities of current sharers. When a request for a block reaches the home, the memory controller looks up this block's directory state. For example, if the request is a GetS, the memory controller looks up the directory state to determine the owner. If the LLC/memory is the owner, the memory controller completes the transaction by sending a data response to the requestor. If a cache controller is the owner, the memory controller forwards the request to the owner cache; when the owner cache receives the forwarded request, it completes the transaction by sending a data response to the requestor.

The choice of snooping versus directory involves making tradeoffs. Snooping protocols are logically simple, but they do not scale to large numbers of cores because broadcasting does not scale.

Directory protocols are scalable because they unicast, but many transactions take more time because they require an extra message to be sent when the home is not the owner. In addition, the choice of protocol affects the interconnection network (e.g., classical snooping protocols require a total order for request messages).

6.4.3.2 Invalidate vs. Update

The other major design decision in a coherence protocol is to decide what to do when a core writes to a block. This decision is independent of whether the protocol is snooping or directory. There are two options:

- Invalidate protocol: When a core wishes to write to a block, it initiates a coherence transaction to invalidate the copies in all other caches. Once the copies are invalidated, the requestor can write to the block without the possibility of another core reading the block's old value. If another core wishes to read the block after its copy has been invalidated, it has to initiate a new coherence transaction to obtain the block, and it will obtain a copy from the core that wrote it, thus preserving coherence.
- Update protocol: When a core wishes to write a block, it initiates a coherence transaction to update the copies in all other caches to reflect the new value it wrote to the block.

Once again, there are tradeoffs involved in making this decision. Update protocols reduce the latency for a core to read a newly written block because the core does not need to initiate and wait for a GetS transaction to complete. However, update protocols typically consume substantially more bandwidth than invalidate protocols because update messages are larger than invalidate messages (an address and a new value, rather than just an address). Furthermore, update protocols greatly complicate the implementation of many memory consistency models. For example, preserving write atomicity (Section 5.5) becomes much more difficult when multiple caches must apply multiple updates to multiple copies of a block. Because of the complexity of update protocols, they are rarely implemented; in this primer, we focus on the far more common invalidate protocols.

6.4.3.3 Hybrid Designs

For both major design decisions, one option is to develop a hybrid. There are protocols that combine aspects of snooping and directory protocols [6, 2], and there are protocols that combine aspects of invalidate and update protocols [8]. The design space is rich and architects are not constrained to following any particular style of design.

6.5 REFERENCES

[1] A. Charlesworth. The Sun Fireplane SMP Interconnect in the Sun 6800. In *Proceedings of 9th Hot Interconnects Symposium*, Aug. 2001. doi:10.1109/HIS.2001.946691

[2] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, March/April 2007. doi:10.1109/MM.2007.43

[3] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-access Times. *Electronics*, 57(1):164–169, Jan. 1984.

[4] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, May 1981.

[5] H. Q. Le et al. IBM POWER6 Microarchitecture. *IBM Journal of Research and Development*, 51(6), 2007. doi:10.1147/rd.516.0639

[6] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pp. 251–262, Jan. 2002. doi:10.1109/HPCA.2002.995715

[7] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, volume I, pp. 1–10, Aug. 1995.

[8] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocesors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996. doi:10.1109/HPCA.1996.501197

[9] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002. doi:10.1109/TPDS.2002.1011412

[10] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 414–423, June 1986.

CHAPTER 7

Snooping Coherence Protocols

In this chapter, we present snooping coherence protocols. Snooping protocols were the first widely-deployed class of protocols and they continue to be used in a variety of systems. Snooping protocols offer many attractive features, including low-latency coherence transactions and a conceptually simpler design than the alternative, directory protocols (Chapter 8).

We first introduce snooping protocols at a high level (Section 7.1). We then present a simple system with a complete but unsophisticated three-state (MSI) snooping protocol (Section 7.2). This system and protocol serve as a baseline upon which we later add system features and protocol optimizations. The protocol optimizations that we discuss include the additions of the Exclusive state (Section 7.3) and the Owned state (Section 7.4), as well as higher performance interconnection networks (Sections 7.5 and 7.6). We then discuss commercial systems with snooping protocols (Section 7.7) before concluding the chapter with a discussion of snooping and its future (Section 7.8).

Given that some readers may not wish to delve too deeply into snooping, we have organized the chapter such that readers may skim or skip Sections 7.3 through 7.6, if they so choose.

7.1 INTRODUCTION TO SNOOPING

Snooping protocols are based on one idea: all coherence controllers observe (*snoop*) coherence requests in the same order and collectively "do the right thing" to maintain coherence. By requiring that all requests to a given block arrive in order, a snooping system enables the distributed coherence controllers to correctly update the finite state machines that collectively represent a cache block's state.

Traditional snooping protocols broadcast requests to all coherence controllers, including the controller that initiated the request. The coherence requests typically travel on an ordered broadcast network, such as a bus. The ordered broadcast ensures that every coherence controller observes the same series of coherence requests in the same order, i.e., that there is a total order of coherence requests. Since a total order subsumes all per-block orders, this total order guarantees that all coherence controllers can correctly update a cache block's state.

TABLE 7.1: Snooping Coherence Example. All Activity Involves Block A (Denoted "A:")

time	Core C1	Core C2	LLC/Memory
0	A:I	A:I	A:I (LLC/memory is owner)
1	A:GetM from Core C1 / M	A:GetM from Core C1 / I	A:GetM from Core C1 / M (LLC/memory is not owner)
2	A:GetM from Core C2 / I	A:GetM from Core C2 / M	A: GetM from Core C2 / M

To illustrate the importance of processing coherence requests in the same per-block order, consider the examples in Tables 7.1 and 7.2 where both core C1 and core C2 want to get the same block A in state M. In Table 7.1, all three coherence controllers observe the same per-block order of coherence requests and collectively maintain the single-writer–multiple-reader (SWMR) invariant. Ownership of the block progresses from the LLC/memory to core C1 to core C2. Every coherence controller independently arrives at the correct conclusion about the block's state as a result of each observed request. Conversely, Table 7.2 illustrates how incoherence might arise if core C2 observes a different per-block order of requests than core C1 and the LLC/memory. First, we have a situation in which both core C1 and core C2 are simultaneously in state M, which violates the SWMR invariant. Next, we have a situation in which no coherence controller believes it is the owner and thus a coherence request at this time would not receive a response (perhaps resulting in deadlock).

Traditional snooping protocols create a total order of coherence requests across *all blocks*, even though coherence requires only a per-block order of requests. Having a total order makes it easier

TABLE 7.2: Snooping (In)Coherence Example. All Activity Involves Block A (Denoted "A:")

time	Core C1	Core C2	LLC/Memory
0	A:I	A:I	A:I (LLC/memory is owner)
1	A: GetM from Core C1 / M	**A: GetM from Core C2** / M	A: GetM from Core C1 / M (LLC/memory is not owner)
2	A: GetM from Core C2 / I	**A: GetM from Core C1** / I	A: GetM from Core C2 / M

TABLE 7.3: Per-block Order, Coherence, and Consistency. States and Operations that Pertain to Address A are Preceeded by the Prefix "A:", and we Denote a Block A in State X with Value V as "A:X[V]". If the Value is Stale, We Omit It (e.g., "A:I").

time	Core C1	Core C2	LLC/Memory
0	A:I B:M[0]	A:S[0] B:I	A:S[0] B:M
1	A: GetM from Core C1 / M[0] **store A = 1** B:M[0]	A:S[0] B:I	A:S[0] B:M
2	A:M[1] **store B = 1** B:M[1]	A:S[0] B:I	A:GetM from Core C1 / M B:M
3	A:M[1] B:GetS from Core C2 / S[1]	A:S[0] B:I	A:M B:GetS from Core C2 / S[1]
4	A:M[1] B:S[1]	A:S[0] B:GetS from Core C2/S[1] **r1 = B[1]**	A:M B:S[1]
5	A:M[1] B:S[1]	A:S[0] **r2 = A[0]** B:S[1]	A:M B:S[1]
6	A:M[1] B:S[1]	A: GetM from Core1 / I B:S[1]	A:M B:S[1]
r1 = 1, r2 = 0 violates SC and TSO			

to implement memory consistency models that require a total order of memory references, such as SC and TSO. Consider the example in Table 7.3 which involves two blocks A and B; each block is requested exactly once and so the system trivially observes per-block request orders. Yet because cores C1 and C2 observe the GetM and GetS requests out-of-order, this execution violates both the SC and TSO memory consistency models.

Sidebar: How Snooping Depends on a Total Order of Coherence Requests

At first glance, the reader may assume that the problem in Table 7.3 arises because the SWMR invariant is violated for block A in cycle 1, since C1 has an M copy and C2 still has an S copy. However, Table 7.4 illustrates the same example, but enforces a total order of coherence requests. This example is identical until cycle 4, and thus has the same apparent SWMR violation. However, like the proverbial "tree in the forest," this violation does not cause a problem because it is not observed (i.e., there is "no one there to hear it"). Specifically, because the cores see both requests in the same order, C2 invalidates block A before it can see the new value for block B. Thus when C2 reads block A, it must get the new value and therefore yields a correct SC and TSO execution.

Traditional snooping protocols use the total order of coherence requests to determine when, in a logical time based on snoop order, a particular request has been observed. In the example of Table 7.4, because of the total order, core C1 can infer that C2 will see the GetM for A before the GetS for B, and thus C2 does not need to send a specific acknowledgement message when it receives the coherence message. This implicit acknowledgment of request reception distinguishes snooping protocols from the directory protocols we study in the next chapter.

TABLE 7.4. Total Order, Coherence, and Consistency. States and Operations that Pertain to Address A are Preceded by the Prefix "A:", and we Denote a Block A in State X with Value V as "A:X[V]". If the Value is Stale, We Omit It (e.g., "A:I").

time	Core C1	Core C2	LLC/Memory
0	A:I B:M[0]	A:S[0] B:I	A:S[0] B:M
1	A: GetM from Core C1 / M[0] **store A = 1** B:M[0]	A:S[0] B:I	A:S[0] B:M
2	A:M[1] **store B = 1** B:M[1]	A:S[0] B:I	A: GetM from Core C1 / M B:M
3	A:M[1] B: GetS from Core C2 / S[1]	A:S[0] B:I	A:M B:GetS from Core C2 / S[1]
4	A:M[1] B:S[1]	A: GetM from Core1 / I B:I	A:M B:S[1]
5	A:M[1] B:S[1]	A:I B: GetS from Core C2/S[1] **r1 = B[1]**	A:M B:S[1]
6	A: GetS from Core C2/S[1] B:S[1]	A: GetS from Core C2/S[1] **r2 = A[1]** B:S[1]	A:GetS from Core C2 / S[1] B:S[1]
r1 = 1, r2 = 1 satisfies SC and TSO			

We discuss some of the subtler issues regarding the need for a total order in the sidebar.

Requiring that broadcast coherence requests be observed in a total order has important implications for the interconnection network used to implement traditional snooping protocols. Because many coherence controllers may simultaneously attempt to issue coherence requests, the interconnection network must serialize these requests into some total order. However the network determines this order, this mechanism becomes known as the protocol's *serialization (ordering) point*. In the general case, a coherence controller issues a coherence request, the network orders that request at the serialization point and broadcasts it to all controllers, and the issuing controller learns where its request has been ordered by snooping the stream of requests it receives from the controller. As a concrete and simple example, consider a system which uses a bus to broadcast coherence requests. Coherence controllers must use arbitration logic to ensure that only a single request is issued on the bus at once. This arbitration logic acts as the serialization point because it effectively determines the order in which requests appear on the bus. A subtle but important point is that a coherence request is ordered the instant the arbitration logic serializes it, but a controller may only be able to determine this order by snooping the bus to observe which other requests appear before and after its own request. Thus, coherence controllers may observe the total request order several cycles after the serialization point determines it.

Thus far, we have discussed only coherence requests, but not the responses to these requests. The reason for this seeming oversight is that the key aspects of snooping protocols revolve around the requests. There are few constraints on response messages. They can travel on a separate interconnection network that does not need to support broadcast nor have any ordering requirements. Because response messages carry data and are thus much longer than requests, there are significant benefits to being able to send them on a simpler, lower-cost network. Notably, response messages do not affect the serialization of coherence transactions. Logically, a coherence transaction—which consists of a broadcast request and a unicast response—occurs when the request is ordered, *regardless of when the response arrives at the requestor*. The time interval between when the request appears on the bus and when the response arrives at the requestor does affect the implementation of the protocol (e.g., during this gap, are other controllers allowed to request this block? If so, how does the requestor respond?), but it does not affect the serialization of the transaction.[1]

7.2 BASELINE SNOOPING PROTOCOL

In this section, we present a straightforward, unoptimized snooping protocol and describe its implementation on two different system models. The first, simple system model illustrates the basic

[1]This logical serialization of coherence transactions is analogous to the logical serialization of instruction execution in processor cores. Even when a core performs out-of-order execution, it still commits (serializes) instructions in program order.

approach for implementing snooping coherence protocols. The second, modestly more complex baseline system model illustrates how even relatively simple performance improvements may impact coherence protocol complexity. These examples provide insight into the key features of snooping protocols while revealing inefficiencies that motivate the features and optimizations presented in subsequent sections of this chapter. Sections 7.5 and 7.6 discuss how to adapt this baseline protocol for more advanced system models.

7.2.1 High-Level Protocol Specification

The baseline protocol has only three stable states: M, S, and I. Such a protocol is typically referred to as an MSI protocol. Like the protocol in Section 6.3, this protocol assumes a write-back cache. A block is owned by the LLC/memory unless the block is in a cache in state M. Before presenting the detailed specification, we first illustrate a higher level abstraction of the protocol in order to understand its fundamental behaviors. In Figures 7.1 and 7.2, we show the transitions between the stable states at the cache and memory controllers, respectively.

There are three notational issues to be aware of. First, in Figure 7.1, the arcs are labeled with coherence requests that are observed on the bus. We intentionally omit other events, including loads, stores, and coherence responses. Second, the coherence events at the cache controller are labeled with either "Own" or "Other" to denote whether the cache controller observing the request is the requestor or not. Third, in Figure 7.2, we specify the state of a block at memory using a cache-centric notation (e.g., a memory state of M denotes that there exists a cache with the block in state M).

7.2.2 Simple Snooping System Model: Atomic Requests, Atomic Transactions

Figure 7.3 illustrates the simple system model, which is nearly identical to the baseline system model introduced in Figure 2.1. The only difference is that the generic interconnection network from Figure 2.1 has been specified as a bus. Each core can issue load and store requests to its cache controller; the cache controller will choose a block to evict when it needs to make room for another block. The bus facilitates a total order of coherence requests that are snooped by all coherence controllers. Like the example in the previous chapter, this system model has atomicity properties that simplify the coherence protocol. Specifically, this system implements two atomicity properties which we define as *Atomic Requests* and *Atomic Transactions*. The *Atomic Requests* property states that a coherence request is ordered in the same cycle that it is issued. This property eliminates the possibility of a block's state changing—due to another core's coherence request—between when a request is issued and when it is ordered. The *Atomic Transactions* property states that coherence transactions are *atomic* in that a subsequent request for the *same* block may not appear on the bus until after the first transaction completes (i.e., until after the response has appeared on the bus). Because coherence involves operations on a single block, whether or not the system permits subsequent requests

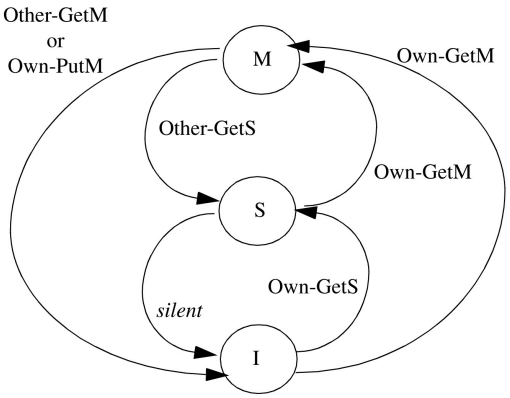

FIGURE 7.1: MSI: Transitions between stable states at cache controller.

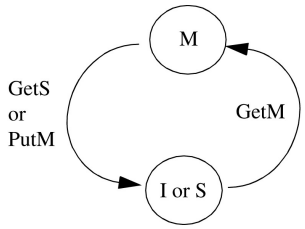

FIGURE 7.2: MSI: Transitions between stable states at memory controller.

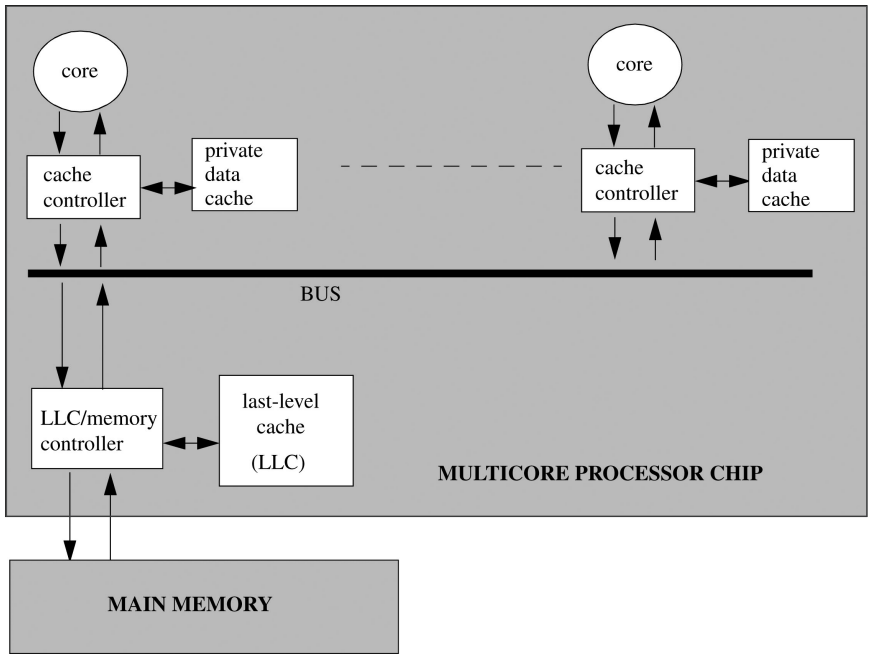

FIGURE 7.3: Simple snooping system mode.

to *different* blocks does not impact the protocol. Although simpler than most current systems, this system model resembles the SGI Challenge, a successful machine in the 1980s [5].

7.2.2.1 Detailed Protocol Specification

Tables 7.5 and 7.6 present the detailed coherence protocol for the simple system model. Compared to the high-level description in Section 7.2.1, the most significant difference is the addition of two transient states in the cache controller and one in the memory controller. This protocol has very few transient states because the atomicity constraints of the simple system model greatly limit the number of possible message interleavings.

> **Flashback to Quiz Question** 6: In an MSI snooping protocol, a cache block may only be in one of three coherence states. *True or false?*
> **Answer**: *False!* Even for the simplest system model, there are more than three states, because of transient states.

Shaded entries in the table denote impossible (or at least erroneous) transitions. For example, a cache controller should never receive a *Data* message for a block that it has not requested (i.e., a block in state I in its cache). Similarly, the *Atomic Transactions* constraint prevents another core from issuing a subsequent request before the current transaction completes; the table entries labeled "(A)" cannot occur due to this constraint. Blank entries denote legal transitions that require no action. These tables omit many implementation details that are not necessary for understanding the protocol. Also, in this protocol and the rest of the protocols in this chapter, we omit the event corresponding to Data for another core's transaction; a core never takes any action in response to observing Data on the bus for another core's transaction.

As with all MSI protocols, loads may be performed (i.e., hit) in states S and M, while stores hit only in state M. On load and store misses, the cache controller initiates coherence transactions by sending GetS and GetM requests, respectively.[2] The transient states IS^D, IM^D, and SM^D indicate that the request message has been sent, but the data response (Data) has not yet been received. In these transient states, because the requests have already been ordered, the transactions have already been ordered and the block is *logically* in state S, M, or M, respectively. A load or store must

[2] We do not include an Upgrade transaction in this protocol, which would optimize the S-to-M transition by not needlessly sending data to the requestor. Adding an Upgrade would be fairly straightforward for this system model with *Atomic Requests*, but it is significantly more complicated without *Atomic Requests*. We discuss this issue when we present a protocol without *Atomic Requests*.

TABLE 7.5: Simple Snooping (Atomic Requests, Atomic Transactions): Cache Controller

States	Processor Core Events			Bus Events						
				Own Transaction				Transactions For Other Cores		
	Load	Store	Replacement	Own-GetS	Own-GetM	Own-PutM	Data	Other-GetS	Other-GetM	Other-PutM
I	issue GetS /IS^D	issue GetM /IM^D								
IS^D	stall Load	stall Store	stall Evict				copy data into cache, load hit /S	(A)	(A)	(A)
IM^D	stall Load	stall Store	stall Evict				copy data into cache, store hit /M	(A)	(A)	(A)
S	load hit	issue GetM /SM^D	-/I						-/I	
SM^D	load hit	stall Store	stall Evict				copy data into cache, store hit /M	(A)	(A)	(A)
M	load hit	store hit	issue PutM, send Data to memory /I					send Data to req and memory /S	send Data to req /I	

TABLE 7.6: Simple Snooping (Atomic Requests, Atomic Transactions): Memory Controller

State	Bus Events			
	GetS	GetM	PutM	Data from Owner
IorS	send data block in Data message to requestor/IorS	send data block in Data message to requestor/M		
$IorS^D$	(A)	(A)		update data block in memory/IorS
M	-/$IorS^D$		-/$IorS^D$	

wait for the Data to arrive, though.[3] Once the data response appears on the bus, the cache controller can copy the data block into the cache, transition to stable state S or M, as appropriate, and perform the pending load or store.

The system model's atomicity properties simplify cache miss handling in two ways. First, the *Atomic Requests* property ensures that when a cache controller seeks to upgrade permissions to a block—to go from I to S, I to M, or S to M—it can issue a request without worrying that another core's request might be ordered ahead of its own. Thus, the cache controller can transition immediately to state IS^D, IM^D, or SM^D, as appropriate, to wait for a data response. Similarly, the *Atomic Transactions* property ensures that no subsequent requests for a block will occur until after the current transaction completes, eliminating the need to handle requests from other cores while in one of these transient states.

A data response may come from either the memory controller or another cache that has the block in state M. A cache that has a block in state S can ignore GetS requests because the memory controller is required to respond, but must invalidate the block on GetM requests to enforce the coherence invariant. A cache that has a block in state M must respond to both GetS and GetM requests, sending a data response and transitioning to state S or state I, respectively.

The LLC/memory has two stable states, M and IorS, and one transient state $IorS^D$. In state IorS, the memory controller is the owner and responds to both GetS and GetM requests because this state indicates that no cache has the block in state M. In state M, the memory controller does not respond with data because the cache in state M is the owner and has the most recent copy of the data. However, a GetS in state M means that the cache controller will transition to state S, so the memory controller must also get the data, update memory, and begin responding to all future requests. It does this by transitioning immediately to the transient state $IorS^D$ and waits until it receives the data from the cache that owns it.

When the cache controller evicts a block due to a replacement decision, this leads to the protocol's two possible coherence downgrades: from S to I and from M to I. In this protocol, the S-to-I downgrade is performed "silently" in that the block is evicted from the cache without any communication with the other coherence controllers. In general, silent state transitions are possible only when all other coherence controllers' behavior remains unchanged; for example, a silent eviction of an owned block is not allowable. The M-to-I downgrade requires communication because the M copy of the block is the only valid copy in the system and cannot simply be discarded. Thus, another coherence controller (i.e., the memory controller) must change its state. To replace a block in state M, the cache controller issues a PutM request on the bus and then sends the data back to the memory controller. At the LLC, the block enters state $IorS^D$ when the PutM request arrives, then

[3]Technically, a store may be performed as soon as the request is ordered, so long as the newly stored value is not overwritten when the Data arrives. Similarly, a subsequent load to a newly written value is permitted.

transitions to state IorS when the Data message arrives.[4] The *Atomic Requests* property simplifies the cache controller, by preventing an intervening request that might downgrade the state (e.g., another core's GetM request) before the PutM gets ordered on the bus. Similarly, the *Atomic Transactions* property simplifies the memory controller by preventing other requests for the block until the PutM transaction completes and the memory controller is ready to respond to them.

7.2.2.2 Running Example

In this section, we present an example execution of the system to show how the coherence protocol behaves in a common scenario. We will use this example in subsequent sections both to understand the protocols and also to highlight differences between them. The example includes activity for just one block, and initially, the block is in state I in all caches and in state IorS at the LLC/memory.

In this example, illustrated in Table 7.7, cores C1 and C2 issue load and store instructions, respectively, that miss on the same block. core C1 attempts to issue a GetS and core C2 attempts to issue a GetM. We assume that core C1's request happens to get serialized first and the *Atomic Transactions* property prevents core C2's request from reaching the bus until C1's request completes. The memory controller responds to C1 to complete the transaction on cycle 3. Then, core C2's GetM is serialized on the bus; C1 invalidates its copy and the memory controller responds to C2 to complete that transaction. Lastly, C1 issues another GetS. C2, the owner, responds with the data and changes its state to S. C2 also sends a copy of the data to the memory controller because the LLC/memory is now the owner and needs an up-to-date copy of the block. At the end of this execution, C1 and C2 are in state S and the LLC/memory is in state IorS.

7.2.3 Baseline Snooping System Model: Non-Atomic Requests, Atomic Transactions

The baseline snooping system model, which we use for most of the rest of this chapter, differs from the simple snooping system model by permitting non-atomic requests. Non-atomic requests arise from a number of implementation optimizations, but most commonly due to inserting a message queue (or even a single buffer) between the cache controller and the bus. By separating when a request is issued from when it is ordered, the protocol must address a window of vulnerability that did not exist in the simple snooping system. The baseline snooping system model preserves the *Atomic Transactions* property, which we do not relax until Section 7.5.

We present the detailed protocol specification, including all transient states, in Tables 7.8 and 7.9. Compared to the protocol for the simple snooping system in Section 7.2.2, the most significant difference is the much larger number of transient states. Relaxing the *Atomic Requests* property

[4] We make the simplifying assumption that these messages cannot arrive out of order at the memory controller.

TABLE 7.7: Simple Snooping: Example Execution. All Activity is for One Block.

cycle	Core C1	Core C2	LLC/memory	request on bus	data on bus
Initial	I	I	IorS		
1	load miss; issue GetS / IS^D				
2				GetS (C1)	
3		store miss; stall due to *Atomic Transactions*	send response to C1		
4					data from LLC/mem
5	copy data to cache; perform load / S	issue GetM / IM^D			
6				GetM (C2)	
7	- / I		send response to C2 / M		
8					data from LLC/mem
9		copy data to cache; perform store / M			
10	load miss; issue GetS / IS^D				
11				GetS (C1)	
12		send data to C1 and to LLC/mem / S	- / $IorS^D$		
13					data from C2
14	copy data from C2; perform load / S		copy data from C2 / IorS		

introduces numerous situations in which a cache controller observes a request from another controller on the bus in between issuing its coherence request and observing its own coherence request on the bus.

Taking the I-to-S transition as an example, the cache controller issues a GetS request and changes the block's state from I to IS^{AD}. Until the requesting cache controller's own GetS is observed on the bus and serialized, the block's state is *effectively* I. That is, the requestor's block is treated as if it were in I; loads and stores cannot be performed and coherence requests from other nodes must be ignored. Once the requestor observes its own GetS, the request is ordered and block is logically S, but loads cannot be performed because the data has not yet arrived. The cache controller changes the block's state to IS^D and waits for the data response from the previous owner. Because

TABLE 7.8: MSI Snooping Protocol with Atomic Transactions-Cache Controller. A Shaded Entry Labeled "(A)" Denotes that this Transition is Impossible Because Transactions are Atomic on Bus.

	load	store	replacement	OwnGetS	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own Data response
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-	
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-	
IS^{D}	stall	stall	stall				(A)	(A)		-/S
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-	
IM^{D}	stall	stall	stall				(A)	(A)		-/M
S	hit	issue GetM/SM^{AD}	-/I				-	-/I	-	
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}	-	
SM^{D}	hit	stall	stall				(A)	(A)		-/M
M	hit	hit	issue PutM/MI^{A}				send data to requestor and to memory/S	send data to requestor/I	-	
MI^{A}	hit	hit	stall			send data to memory/I	send data to requestor and to memory/II^{A}	send data to requestor/II^{A}		
II^{A}	stall	stall	stall			send NoData to memory/I	-	-		

TABLE 7.9: MSI Snooping Protocol with Atomic Transactions - Memory Controller. A Shaded Entry Labeled "(A)" Denotes that this Transition is Impossible Because Transactions are Atomic on Bus.

	GetS	GetM	PutM	Data From Owner	NoData
IorS	send data to requestor	send data to requestor/M	-/$IorS^{D}$		
$IorS^{D}$	(A)	(A)		write data to LLC/memory /IorS	-/IorS
M	-/$IorS^{D}$		-/M^{D}		
M^{D}	(A)	(A)		write data to LLC/IorS	-/M

of the *Atomic Transactions* property, the data message is the next coherence message (to the same block). Once the data response arrives, the transaction is complete and the requestor changes the block's state to the stable S state and performs the load. The I-to-M transition proceeds similarly to this I-to-S transition.

The transition from S to M illustrates the potential for state changes to occur during the window of vulnerability. If a core attempts to store to a block in state S, the cache controller issues a GetM request and transitions to state SM^{AD}. The block remains effectively in state S, so loads may continue to hit and the controller ignores GetS requests from other cores. However, if another core's GetM request gets ordered first, the cache controller must transition the state to IM^{AD} to prevent further load hits. The window of vulnerability during the S-to-M transition complicates the addition of an Upgrade transaction, as we discuss in the sidebar.

Sidebar: Upgrade Transactions in Systems Without Atomic Requests

For the protocol with *Atomic Requests*, an Upgrade transaction is an efficient way for a cache to transition from Shared to Modified. The Upgrade request invalidates all shared copies, and it is much faster than issuing a GetM, because the requestor needs to wait only until the Upgrade is serialized (i.e., the bus arbitration latency) rather than wait for data to arrive from the LLC/memory.

However, without *Atomic Requests*, adding an Upgrade transaction becomes more difficult because of the window of vulnerability between issuing a request and when the request is serialized. The requestor may lose its shared copy due to an Other-GetM or Other-Upgrade that is serialized during this window of vulnerability. The simplest solution to this problem is to change the block's state to a new state in which it waits for its own Upgrade to be serialized. When its Upgrade is serialized, which will invalidate other S copies (if any) but will not return data, the core must then issue a subsequent GetM request to transition to M.

Handling Upgrades more efficiently is difficult, because the LLC/memory needs to know when to send data. Consider the case in which cores C0 and C2 have a block A shared and both seek to upgrade it and, at the same time, core C1 seeks to read it. C0 and C2 issue Upgrade requests and C1 issues a GetS request. Suppose they serialize on the bus as C0, C1, and C2. C0's Upgrade succeeds, so the LLC/memory (in state IorS) should change its state to M but not send any data, and C2 should invalidate its S copy. C1's GetS finds the block in state M at C0, which responds with the new data value and updates the LLC/memory back to state IorS. C2's Upgrade finally appears, but because it has lost its shared copy, it needs the LLC/memory to respond. Unfortunately, the LLC/memory is in state IorS and cannot tell that this Upgrade needs data. Alternatives exist to solve this issue, but are outside the scope of this primer.

The window of vulnerability also affects the M-to-I coherence downgrade, in a much more significant way. To replace a block in state M, the cache controller issues a PutM request and changes the block state to MI^A; unlike the protocol in Section 7.2.2, it does not immediately send the data to the memory controller. Until the PutM is observed on the bus, the block's state is effectively M and the cache controller must respond to other cores' coherence requests for the block. In the case where no intervening coherence requests arrive, the cache controller responds to observing its own PutM by sending the data to the memory controller and changing the block state to state I. If an intervening GetS or GetM request arrives before the PutM is ordered, the cache controller must respond as if it were in state M and then transition to state II^A to wait for its PutM to appear on the bus. Intuitively, the cache controller should simply transition to state I once it sees its PutM because it has already given up ownership of the block. Unfortunately, doing so will leave the memory controller stuck in a transient state because it also receives the PutM request. Nor can the cache controller simply send the data anyway because doing so might overwrite valid data.[5] The solution is for the cache controller to send a special NoData message to the memory controller when it sees its PutM while in state II^A. The memory controller is further complicated by needing to know which stable state it should return to if it receives a NoData message. We solve this problem by adding a second transient memory state M^D. Note that these transient states represent an exception to our usual transient state naming convention. In this case, state X^D indicates that the memory controller should revert to state X when it receives a NoData message (and move to state IorS if it receives a data message).

7.2.4 Running Example

Returning to the running example, illustrated in Table 7.10, core C1 issues a GetS and core C2 issues a GetM. Unlike the previous example (in Table 7.7), eliminating the *Atomic Requests* property means that both cores issue their requests and change their state. We assume that core C1's request happens to get serialized first, and the *Atomic Transactions* property ensures that C2's request does not appear on the bus until C1's transaction completes. After the LLC/memory responds to complete C1's transaction, core C2's GetM is serialized on the bus. C1 invalidates its copy and the LLC/memory responds to C2 to complete that transaction. Lastly, C1 issues another GetS. When this GetS reaches the bus, C2, the owner, responds with the data and changes its state to S. C2 also sends a copy of the data to the memory controller because the LLC/memory is now the owner and

[5] Consider the case in which core C1 has a block in M and issues a PutM, but core C2 does a GetM and core C3 does a GetS, both of which are ordered before C1's PutM. C2 gets the block in M, modifies the block, and then in response to C3's GetS, updates the LLC/memory with the updated block. When C1's PutM is finally ordered, writing the data back would overwrite C2's update.

TABLE 7.10: Baseline Snooping: Example Execution.

cycle	Core C1	Core C2	LLC/memory	request on bus	data on bus
1	issue GetS / IS^{AD}				
2		issue GetM / IM^{AD}			
3				**GetS (C1)**	
4	- / IS^{D}		send data to C1 /IorS		
5					**data from LLC/mem**
6	copy data from LLC/mem / S			**GetM (C2)**	
7	- / I	- / IM^{D}	send data to C2 / M		
8					**data from LLC/mem**
9		copy data from LLC/mem / M			
10	issue GetS / IS^{AD}				
11				**GetS (C1)**	
12	- / IS^{D}	send data to C1 and to LLC/mem / S	- / $IorS^{D}$		
13					**data from C2**
14	copy data from C2 / S		copy data from C2 / IorS		

needs an up-to-date copy of the block. At the end of this execution, C1 and C2 are in state S and the LLC/memory is in state IorS.

7.2.5 Protocol Simplifications

This protocol is relatively straightforward and sacrifices performance to achieve this simplicity. The most significant simplification is the use of atomic transactions on the bus. Having atomic transactions eliminates many possible transitions, denoted by "(A)" in the tables. For example, when a core has a cache block in state IM^{D}, it is not possible for that core to observe a coherence request for that block from another core. If transactions were not atomic, such events could occur and would force us to redesign the protocol to handle them, as we show in Section 7.5.

Another notable simplification that sacrifices performance involves the event of a store request to a cache block in state S. In this protocol, the cache controller issues a GetM and changes the block state to SM^{AD}. A higher performance but more complex solution would use an upgrade transaction, as discussed in the earlier sidebar.

7.3 ADDING THE EXCLUSIVE STATE

There are many important protocol optimizations, which we discuss in the next several sections. More casual readers may want to skip or skim these sections on first reading. One very commonly used optimization is to add the Exclusive (E) state, and in this section, we describe how to create a MESI snooping protocol by augmenting the baseline protocol from Section 7.2.3 with the E state. Recall from Chapter 2 that if a cache has a block in the Exclusive state, then the block is valid, read-only, clean, exclusive (not cached elsewhere), and owned. A cache controller may silently change a cache block's state from E to M without issuing a coherence request.

7.3.1 Motivation

The Exclusive state is used in almost all commercial coherence protocols because it optimizes a common case. Compared to an MSI protocol, a MESI protocol offers an important advantage in the situation in which a core first reads a block and then subsequently writes it. This is a typical sequence of events in many important applications, including single-threaded applications. In an MSI protocol, on a load miss, the cache controller will initiate a GetS transaction to obtain read permission; on the subsequent store, it will then initiate a GetM transaction to obtain write permission. However, a MESI protocol enables the cache controller to obtain the block in state E, instead of S, in the case that the GetS occurs when no other cache has access to the block. Thus, a subsequent store does not require the GetM transaction; the cache controller can silently upgrade the block's state from E to M and allow the core to write to the block. The E state can thus eliminate half of the coherence transactions in this common scenario.

7.3.2 Getting to the Exclusive State

Before explaining how the protocol works, we must first figure out how the issuer of a GetS determines that there are no other sharers and thus that it is safe to go directly to state E instead of state S. There are at least two possible solutions:

- Adding a wired-OR "sharer" signal to bus: when the GetS is ordered on the bus, all cache controllers that share the block assert the "sharer" signal. If the requestor of the GetS observes that the "sharer" signal is asserted, the requestor changes its block state to S; else, the requestor changes its block state to E. The drawback to this solution is having to implement the wired-OR signal. This additional shared wire might not be problematic in this baseline snooping system model that already has a shared wire bus, but it would greatly complicate implementations that do not use shared wire buses (Section 7.6).
- Maintaining extra state at the LLC: an alternative solution is for the LLC to distinguish between states I (no sharers) and S (one or more sharers), which was not needed for the

MSI protocols. In state I, the memory controller responds with data that is specially labeled as being Exclusive; in state S, the memory controller responds with data that is unlabeled. However, maintaining the S state exactly is challenging, since the LLC must detect when the last sharer relinquishes its copy. First, this requires that a cache controller issues a PutS message when it evicts a block in state S. Second, the memory controller must maintain a count of the sharers as part of the state for that block. This is much more complex and bandwidth intensive than our previous protocols, which allowed for silent evictions of blocks in S. A simpler, but less complete, alternative allows the LLC to conservatively track sharers; that is, the memory controller's state S means that there are zero-or-more caches in state S. The cache controller silently replaces blocks in state S, and thus the LLC stays in S even after the last sharer has been replaced. If a block in state M is written back (with a PutM), the state of the LLC block becomes I. This "conservative S" solution forgoes some opportunities to use the E state (i.e., when the last sharer replaces its copy before another core issues a GetM), but it avoids the need for explicit PutS transactions and still captures many important sharing patterns.

In the MESI protocol we present in this section, we choose the most implementable option—maintaining a conservative S state at the LLC—to both avoid the engineering problems associated with implementing wired-OR signals in high-speed buses and avoid explicit PutS transactions.

7.3.3 High-Level Specification of Protocol

In Figures 7.4 and 7.5, we show the transitions between stable states in the MESI protocol. The MESI protocol differs from the baseline MSI protocol at both the cache and LLC/memory. At the cache, a GetS request transitions to S or E, depending upon the state at the LLC/memory when the GetS is ordered. Then, from state E, the block can be silently changed to M. In this protocol, we use a PutM to evict a block in E, instead of using a separate PutE; this decision helps keep the protocol specification concise, and it has no impact on the protocol functionality.

The LLC/memory has one more stable state than in the MSI protocol. The LLC/memory must now distinguish between blocks that are shared by zero or more caches (the conservative S state) and those that are not shared at all (I), instead of merging those into one single state as was done in the MSI protocol.

In this primer, we consider the E state to be an ownership state, which has a significant effect on the protocol. There are, however, protocols that do not consider the E state to be an ownership state, and the sidebar discusses the issues involved in such protocols.

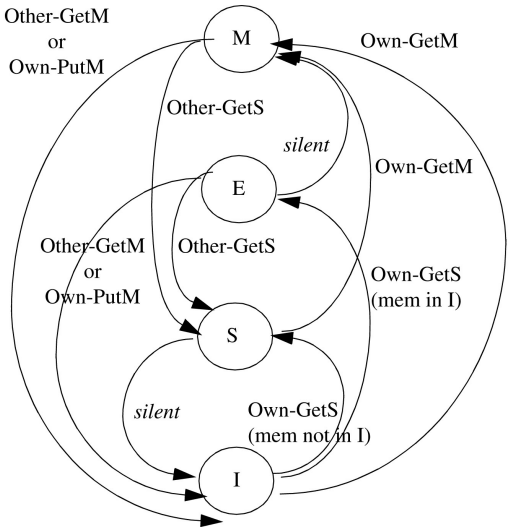

FIGURE 7.4: MESI: Transitions between stable states at cache controller

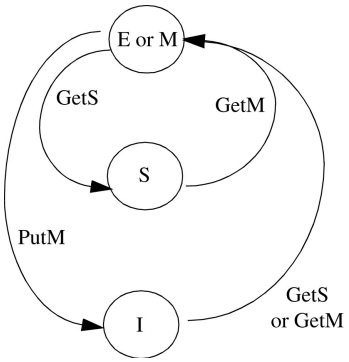

FIGURE 7.5: MESI: Transitions between stable states at memory controller

Sidebar: MESI Snooping if E is Non-ownership State

If the E state is not considered an ownership state (i.e., a block in E is owned by the LLC/memory), then the protocol must figure out which coherence controller should respond to a request after the memory controller has given a block to a cache in state E. Because the transition from state E to state M is silent, the memory controller cannot know whether the cache holds the block in E, in which case the LLC/memory is the owner, or in M, in which case the cache is the owner. If a GetS or GetM is serialized on the bus at this point, the cache can easily determine whether it is the owner and should respond, but the memory controller cannot make this same determination.

One solution to this problem is to have the LLC/memory wait for the cache to respond. When a GetS or GetM is serialized on the bus, a cache with the block in state M responds with data. The memory controller waits a fixed amount of time and, if no response appears in that window of time, the memory controller deduces that it is the owner and that it must respond. If a response from a cache does appear, the memory controller does not respond to the coherence request. This solution has a couple drawbacks, including potentially increased latency for responses from memory. Some implementations hide some or all of this latency by speculatively prefetching the block from memory, at the expense of increased memory bandwidth, power, and energy. A more significant drawback is having to design the system such that the caches' response latency is predictable and short.

7.3.4 Detailed Specification

In Tables 7.11 and 7.12, we present the detailed specification of the MESI protocol, including transient states. Differences with respect to the MSI protocol are highlighted with boldface font. The protocol adds to the set of cache states just the stable E state and the transient state EI^A, but there are several more LLC/memory states, including an extra transient state.

This MESI protocol shares all of the same simplifications present in the baseline MSI protocol. Coherence transactions are still atomic, etc.

TABLE 7.11: MESI Snooping Protocol—Cache Controller. A Shaded Entry Labeled "(A)" Denotes that this Transition is Impossible Because Transactions are Atomic on Bus.

	load	**store**	**replacement**	**OwnGetS**	**OwnGetM**	**OwnPutM**	**OtherGetS**	**OtherGetM**	**OtherPutM**	**Own Data response**	**Own Data response (exclusive)**
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-		
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-		
IS^{D}	stall	stall	stall				(A)	(A)	(A)	-/S	**-/E**
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-		
IM^{D}	stall	stall	stall				(A)	(A)	(A)	-/M	
S	hit	issue GetM/SM^{AD}	-/I				-	-/I	-		
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}	-		
SM^{D}	hit	stall	stall				(A)	(A)	(A)	-/M	
E	**hit**	**hit/M**	**issue PutM/EI^{A}**				**send data to requestor and to memory/S**	**send data to requestor/I**	-		
M	hit	hit	issue PutM/MI^{A}				send data to requestor and to memory/S	send data to requestor/I	-		
MI^{A}	hit	hit	stall			send data to memory/I	send data to requestor and to memory/II^{A}	send data to requestor/II^{A}	-		
EI^{A}	**hit**	**stall**	**stall**			**send NoData-E to memory/I**	**send data to requestor and to memory/II^{A}**	send data to requestor/II^{A}	-		
II^{A}	stall	stall	stall			send NoData to memory/I	-	-	-		

TABLE 7.12: MESI Snooping Protocol—Memory Controller. A Shaded Entry Labeled "(A)" Denotes that this Transition is Impossible Because Transactions are Atomic on Bus.

	GetS	GetM	PutM	Data	NoData	NoData-E
I	send data to requestor/EorM	send data to requestor/EorM	-/I^D			
S	send data to requestor	send data to requestor/EorM	-/S^D			
EorM	-/S^D	-	-/$EorM^D$			
I^D	(A)	(A)	(A)	write data to memory/I	-/I	-/I
S^D	(A)	(A)	(A)	write data to memory/S	-/S	-/S
$EorM^D$	(A)	(A)	(A)	write data to memory/I	-/EorM	-/I

7.3.5 Running Example

We now return to the running example, illustrated in Table 7.13. The execution differs from the MSI protocol almost immediately. When C1's GetS appears on the bus, the LLC/memory is in state I and can thus send C1 Exclusive data. C1 observes the Exclusive data on the bus and changes its state to E (instead of S, as in the MSI protocol). The rest of the execution proceeds similarly to the MSI example, with minor transient state differences.

7.4 ADDING THE OWNED STATE

A second important optimization is the Owned state, and in this section, we describe how to create a MOSI snooping protocol by augmenting the baseline protocol from Section 7.2.3 with the O state. Recall from Chapter 2 that if a cache has a block in the Owned state, then the block is valid, read-only, dirty, and the cache is the owner, i.e., the cache must respond to coherence requests for the block. We maintain the same system model as the baseline snooping MSI protocol; transactions are atomic but requests are not atomic.

7.4.1 Motivation

Compared to an MSI or MESI protocol, adding the O state is advantageous in one specific and important situation: when a cache has a block in state M or E and receives a GetS from another core. In the MSI protocol of Section 7.2.3 and the MESI protocol of Section 7.3, the cache must change the block state from M or E to S and send the data to *both* the requestor *and* the memory controller.

TABLE 7.13: MESI: Example Execution

cycle	Core C1	Core C2	LLC/memory	request on bus	data on bus
1	issue GetS / IS^{AD}				
2		issue GetM / IM^{AD}			
3				GetS (C1)	
4	- / IS^{D}		**send exclusive data to C1 / EorM**		
5					**exclusive data from LLC/mem**
6	**copy data from LLC/mem / E**			**GetM (C2)**	
7	send data to C2/ I	- / IM^{D}	**-/EorM**		
8					data from C1
9		copy data from C1 / M			
10	issue GetS / IS^{AD}				
11				GetS (C1)	
12	- / IS^{D}	send data to C1 and to LLC/mem / S	**- / S^{D}**		
13					data from C2
14	copy data from C2 / S		**copy data from C2 / S**		

The data must be sent to the memory controller because the responding cache relinquishes ownership (by downgrading to state S) and the LLC/memory becomes the owner and thus must thus have an up-to-date copy of the data with which to respond to subsequent requests.

Adding the O state achieves two benefits: (1) it eliminates the extra data message to update the LLC/memory when a cache receives a GetS request in the M (and E) state, and (2) it eliminates the potentially unnecessary write to the LLC (if the block is written again before being written back to the LLC). Historically, for multi-chip multiprocessors, there was a third benefit, which was that the O state allows subsequent requests to be satisfied by the cache instead of by the far-slower memory. Today, in a multicore with an inclusive LLC, as in the system model in this primer, the access latency of the LLC is not nearly as long as that of off-chip DRAM memory. Thus, having a cache respond instead of the LLC is not as big of a benefit as having a cache respond instead of memory.

We now present a MOSI protocol and show how it achieves these two benefits.

7.4.2 High-Level Protocol Specification

We specify a high-level view of the transitions between stable states in Figures 7.6 and 7.7. The key difference is what happens when a cache with a block in state M receives a GetS from another core. In a MOSI protocol, the cache changes the block state to O (instead of S) and retains ownership of the block (instead of transferring ownership to the LLC/memory). Thus, the O state enables the cache to avoid updating the LLC/memory.

7.4.3 Detailed Protocol Specification

In Tables 7.14 and 7.15, we present the detailed specification of the MOSI protocol, including transient states. Differences with respect to the MSI protocol are highlighted with boldface font. The protocol adds two transient cache states in addition to the stable O state. The transient OI^A state helps handle replacements of blocks in state O and the transient OM^A state handles upgrades back to state M after a store. The memory controller has no additional transient states, but we rename what had been the M state to MorO because the memory controller does not need to distinguish between these two states.

To keep the specification as concise as possible, we consolidate the PutM and PutO transactions into a single PutM transaction. That is, a cache evicts a block in state O with a PutM. This

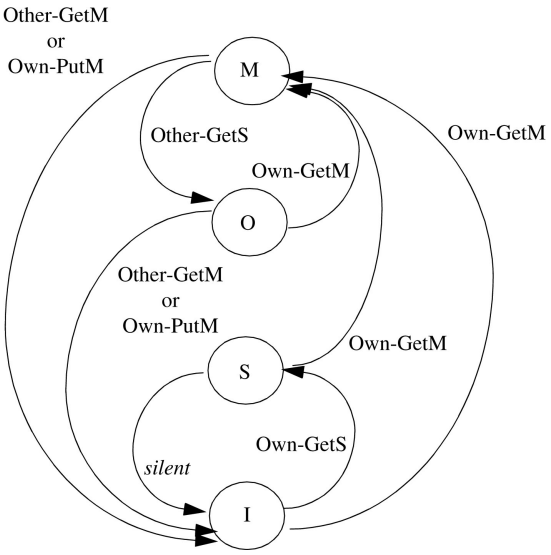

FIGURE 7.6: MOSI: Transitions between stable states at cache controller

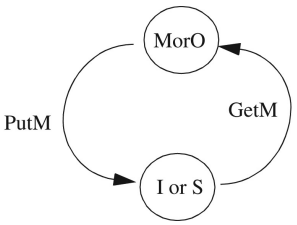

FIGURE 7.7: MOSI: Transitions between stable states at memory controller

TABLE 7.14: MOSI Snooping Protocol—Cache Controller. A Shaded Entry Labeled "(A)" Denotes that this Transition is Impossible Because Transactions are Atomic on Bus.

	load	store	replacement	OwnGetS	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own Data response
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-	
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-	
IS^{D}	stall	stall	stall				(A)	(A)	(A)	-/S
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-	
IM^{D}	stall	stall	stall				(A)	(A)	(A)	-/M
S	hit	issue GetM/SM^{AD}	-/I				-	-/I	-	
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}	-	
SM^{D}	hit	stall	stall				(A)	(A)	(A)	-/M
O	**hit**	**issue GetM /OM^{A}**	**issue PutM/OI^{A}**				**send data to requestor**	**send data to requestor/I**	-	
OM^{A}	**hit**	**stall**	**stall**		**-/M**		**send data to requestor**	**send data to requestor /IM^{AD}**	-	
M	hit	hit	issue PutM/MI^{A}				**send data to requestor/O**	send data to requestor/I	-	
MI^{A}	hit	hit	stall			send data to memory/I	**send data to requestor /OI^{A}**	send data to requestor/II^{A}	-	
OI^{A}	**hit**	**stall**	**stall**			**send data to memory /I**	**send data to requestor**	**send data to requestor/II^{A}**	-	
II^{A}	stall	stall	stall			send NoData to memory/I	-	-	-	

decision has no impact on the protocol's functionality, but does help to keep the tabular specification readable.

This MOSI protocol shares all of the same simplifications present in the baseline MSI protocol. Coherence transactions are still atomic, etc.

7.4.4 Running Example

In Table 7.16, we return to the running example that we introduced for the MSI protocol. The example proceeds identically to the MSI example until C1's second GetS appears on the bus. In the MOSI protocol, this second GetS causes C2 to respond to C1 and change its state to O (instead

TABLE 7.15: MOSI Snooping Protocol—Memory Controller. A Shaded Entry Labeled "(A)" Denotes that this Transition is Impossible Because Transactions are Atomic on Bus.

	GetS	**GetM**	**PutM**	**Data From Owner**	**NoData**
IorS	send data to requestor	send data to requestor/MorO	-/IorS^{D}		
IorS^{D}	(A)	(A)		write data to memory /IorS	-/IorS
MorO	-	-	**-/MorO^{D}**		
MorO^{D}	**(A)**	**(A)**		**write data to memory/IorS**	**-/MorO**

of S). C2 retains ownership of the block and does not need to copy the data back to the LLC/memory (unless and until it evicts the block, not shown).

7.5 NON-ATOMIC BUS

The baseline MSI protocol, as well as the MESI and MOSI variants, all rely on the *Atomic Transactions* assumption. This atomicity greatly simplifies the design of the protocol, but it sacrifices performance.

TABLE 7.16: MOSI: Example Execution.

cycle	**Core C1 (C1)**	**Core C2 (C2)**	**LLC/memory**	**request on bus**	**data on bus**
1	issue GetS / IS^{AD}				
2		issue GetM / IM^{AD}			
3				GetS (C1)	
4	- / IS^{D}		send data to C1 /IorS		
5					data from LLC/mem
6	copy data from LLC/mem / S			GetM (C2)	
7	- / I	- / IM^{D}	**send data to C2 / MorO**		
8					data from LLC/mem
9		copy data from LLC/mem / M			
10	issue GetS / IS^{AD}				
11				GetS (C1)	
12	- / IS^{D}	**send data to C1/ O**	**- / MorO**		
13					data from C2
14	copy data from C2 / S				

7.5.1 Motivation

The simplest way to implement atomic transactions is to use a shared-wire bus with an atomic bus protocol; that is, all bus transactions consist of an indivisible request-response pair. Having an atomic bus is analogous to having an unpipelined processor core; there is no way to overlap activities that could proceed in parallel. Figure 7.8 illustrates the operation of an atomic bus. Because a coherence transaction occupies the bus until the response completes, an atomic bus trivially implements atomic transactions. However, the throughput of the bus is limited by the sum of the latencies for a request and response (including any wait cycles between request and response, not shown). Considering that a response could be provided by off-chip memory, this latency bottlenecks bus performance.

Figure 7.9 illustrates the operation of a pipelined, non-atomic bus. The key advantage is not having to wait for a response before a subsequent request can be serialized on the bus, and thus the bus can achieve much higher bandwidth using the same set of shared wires. However, implementing atomic transactions becomes much more difficult (but not impossible). The atomic transactions property restricts concurrent transactions to the same block, but not different blocks. The SGI Challenge enforced atomic transactions on a pipelined bus using a fast table lookup to check whether or not another transaction was already pending for the same block.

7.5.2 In-Order vs. Out-of-order Responses

One major design issue for a non-atomic bus is whether it is pipelined or split-transaction. A *pipelined* bus, as illustrated in Figure 7.9, provides responses in the same order as the requests. A *split-transaction* bus, illustrated in Figure 7.10, can provide responses in an order different from the request order.

The advantage of a split-transaction bus, with respect to a pipelined bus, is that a low-latency response does not have to wait for a long-latency response to a prior request. For example, if Request 1 is for a block owned by memory and not present in the LLC and Request 2 is for a block owned by an on-chip cache, then forcing Response 2 to wait for Response 1, as a pipelined bus would require, incurs a performance penalty.

One issue raised by a split-transaction bus is matching responses with requests. With an atomic bus, it is obvious that a response corresponds to the most recent request. With a pipelined bus, the requestor must keep track of the number of outstanding requests to determine which message is the response to its request. With a split-transaction bus, the response must carry the identity of the request or the requestor.

7.5.3 Non-Atomic System Model

We assume a system like the one illustrated in Figure 7.11. The request bus and the response bus are split and operate independently. Each coherence controller has connections to and from both

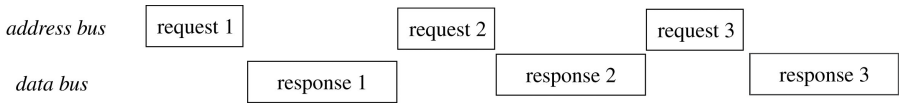

FIGURE 7.8: Atomic bus

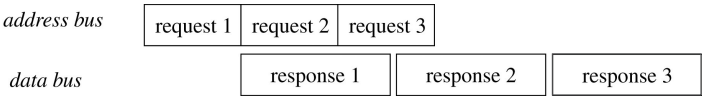

FIGURE 7.9: Pipelined (non-atomic) bus

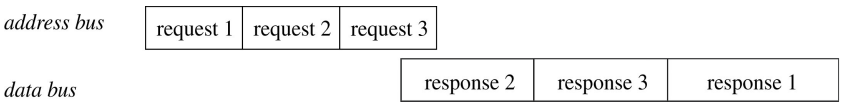

FIGURE 7.10: Split transaction (non-atomic) bus

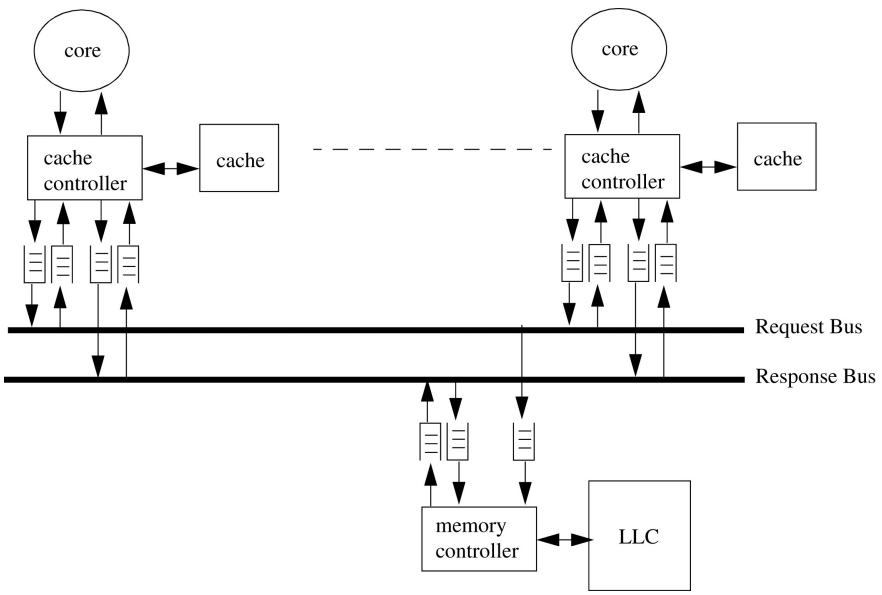

FIGURE 7.11: System model with split-transaction bus

buses, with the exception that the memory controller does not have a connection to make requests. We draw the FIFO queues for buffering incoming and outgoing messages because it is important to consider them in the coherence protocol. Notably, if a coherence controller stalls when processing an incoming request from the request bus, then all requests behind it (serialized after the stalled request) will not be processed by that coherence controller until it processes the currently stalled request. These queues are processed in a strict FIFO fashion, regardless of message type or address.

7.5.4 An MSI Protocol with a Split-Transaction Bus

In this section, we modify the baseline MSI protocol for use in a system with a split-transaction bus. Having a split-transaction bus does not change the transitions between stable states, but it has a large impact on the detailed implementation. In particular, there are many more possible transitions.

In Tables 7.17 and 7.18, we specify the protocol. Several transitions are now possible that were not possible with the atomic bus. For example, a cache can now receive an Other-GetS for a block it has in state IS^D. All of these newly possible transitions are for blocks in transient states in which the cache is awaiting a data response; while waiting for the data, the cache first observes another coherence request for the block. Recall from Section 7.1 that a transaction is ordered based on when its request is ordered on the bus, not when the data arrives at the requestor. Thus, in each of these newly possible transitions, the cache has already effectively completed its transaction but just happens to not have the data yet. Returning to our example of IS^D, the cache block is effectively in S. Thus, the arrival of an Other-GetS in this state requires no action to be taken because a cache with a block in S need not respond to an Other-GetS.

The newly possible transitions other than the above example, however, are more complicated. Consider a block in a cache in state IM^D when an Other-GetS is observed on the bus. The cache block is effectively in state M and the cache is thus the owner of the block but does not yet have the block's data. Because the cache is the owner, the cache must respond to the Other-GetS, yet the cache cannot respond until it receives the data. The simplest solution to this situation is for the cache to stall processing of the Other-GetS until the data response arrives for its Own-GetM. At that point, the cache block will change to state M and the cache will have valid data to send to the requestor of the Other-GetS.

For the other newly possible transitions, at both the cache controller and the memory controller, we also choose to stall until data arrives to satisfy the in-flight request. This is the simplest approach, but it raises three issues. First, it sacrifices some performance, as we discuss in the nextww section.

TABLE 7.17: MSI Snooping Protocol with Split-Transaction Bus—Cache Controller

	load	store	replacement	OwnGetS or OwnGetM	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own Data response (for own request)
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-	
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-	**-/IS^{A}**
IS^{D}	stall	stall	stall				**-**	**stall**		load hit/S
IS^{A}	**stall**	**stall**	**stall**	**load hit/S**			**-**	**-**		
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-	**-/IM^{A}**
IM^{D}	stall	stall	stall				**stall**	**stall**		store hit/M
IM^{A}	**stall**	**stall**	**stall**		**store hit/M**		**-**	**-**		
S	hit	issue GetM/SM^{AD}	-/I				-	-/I		
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}		**-/SM^{A}**
SM^{D}	hit	stall	stall				**stall**	**stall**		store hit/M
SM^{A}	**hit**	**stall**	**stall**		**store hit/M**		**-**	**-/IM^{A}**		
M	hit	hit	issue PutM/MI^{A}				send data to requestor and to memory/S	send data to requestor/I		
MI^{A}	hit	hit	stall			send data to requestor /I	send data to requestor and to memory/II^{A}	send data to requestor/II^{A}		
II^{A}	stall	stall	stall			-/I	-	-	-	

Second, stalling raises the potential of deadlock. If a controller can stall on a message while awaiting another event (message arrival), the architect must ensure that the awaited event will eventually occur. Circular chains of stalls can lead to deadlock and must be avoided. In our protocol in this section, controllers that stall are guaranteed to receive the messages that un-stall them. This guarantee is easy to see because the controller has already seen its own request, the stall only affects the request network, and the controller is waiting for a Data message on the response network.

TABLE 7.18: MSI Snooping Protocol with Split-Transaction Bus—Memory Controller

	GetS	GetM	PutM from Owner	PutM from Non-Owner	Data
IorS	send data to requestor	send data to requestor, set Owner to requestor/M		-	
M	clear Owner/IorS^{D}	set Owner to requestor	clear Owner/IorS^{D}	-	**write data to memory/IorS^{A}**
IorS^{D}	**stall**	**stall**	**stall**	-	write data to memory/IorS
IorS^{A}	**clear Owner/IorS**	-	**clear Owner/IorS**	-	

The third issue raised by stalling coherence requests is that, perhaps surprisingly, it enables a requestor to observe a response to its request before processing its own request. Consider the example in Table 7.19. Core C1 issues a GetM for block X and changes the state of X to IM^{AD}. C1 observes its GetM on the bus and changes state to IM^{D}. The LLC/memory is the owner of X and takes a long time to retrieve the data from memory and put it on the bus. In the meanwhile, core C2 issues a GetM for X that gets serialized on the bus but cannot be processed by C1 (i.e., C1 stalls). C1 issues a GetM for block Y that then gets serialized on the bus. This GetM for Y is queued up behind the previously stalled coherence request at C1 (the GetM from C2) and thus C1 cannot process its own GetM for Y. However, the owner, C2, can process this GetM for Y and responds quickly to C1. Thus, C1 can observe the response to its GetM for Y before processing its request. This possibility requires the addition of transient states. In this example, core C1 changes the state of block Y from IM^{AD} to IM^{A}. Similarly, the protocol also needs to add transient states IS^{A} and SM^{A}. In these transient states, in which the response is observed before the request, the block is effectively in the prior state. For example, a block in IM^{A} is logically in state I because the GetM has not been processed yet; the cache controller does not respond to an observed GetS or GetM if the block is in IM^{A}. We contrast IM^{A} with IM^{D}—in IM^{D}, the block is logically in M and the cache controller must respond to observed GetS or GetM requests once data arrives.

This protocol has one other difference with respect to the previous protocols in this chapter, and the difference pertains to PutM transactions. The situation that is handled differently is when a core, say, core C1, issues a PutM, and a GetS or GetM from another core for the same block gets ordered before C1's PutM. C1 transitions from state MI^{A} to II^{A} before it observes its own PutM. In the atomic protocols earlier in this chapter, C1 observes its own PutM and sends a NoData message to the LLC/memory. The NoData message informs the LLC/memory that the PutM transaction is complete (i.e., it does not have to wait for data). C1 cannot send a Data message to the LLC/memory in this situation because C1's data are stale and the protocol cannot send the LLC/memory stale data that would then overwrite the up-to-date value of the data. In the non-atomic protocols

TABLE 7.19: Example: Response Before Request. Initially, Block X is in State I in Both Caches and Block Y is in State M at Core C2.

cycle	Core C1	Core C2	LLC/memory	request on bus	data on bus
initial	X:I Y:I	X:I Y:M	X:I Y:M		
1	X: store miss; issue GetM/ IM^{AD}				
2				X: GetM (C1)	
3	X: process GetM (C1) / IM^{D}	X: process GetM (C1) - ignore;	X: process GetM (C1) - LLC miss, start accessing X from DRAM		
4		X:store miss; issue GetM/ IM^{AD}			
5	Y: store miss; issue GetM/ IM^{AD}			X: GetM (C2)	
6	X: stall on GetM (C2)	X: process GetM (C2) / IM^{D}	X: process GetM (C2) - ignore	Y: GetM (C1)	
7	Y: process GetM/ IM^{D}	Y: process GetM (C1) - send data to C1 / I	Y: process GetM (C1) - ignore		
8					Y: data from C2
9	**Y: write data into cache /IM^{A}**				
10			X: LLC miss completes, send data to C1		
11					X: data from LLC
12	X: write data into cache/ M Perform store				
13	X: (unstall) process GetM (C2) - send data to C2 / I				
14	**Y: process (in-order) GetM (C1) / M ; perform store**				X: data from C1
15		X: write data into cache/M perform store			

in this chapter, we augment the state of each block in the LLC with a field that holds the identity of the current owner of the block. The LLC updates the owner field of a block on every transaction that changes the block's ownership. Using the owner field, the LLC can identify situations in which a PutM from a non-owner is ordered on the bus; this is exactly the same situation in which C1 is in state II^{A} when it observes its PutM. Thus, the LLC knows what happened and C1 does not have to send a NoData message to the LLC. We chose to modify how PutM transactions are handled in the non-atomic protocols, compared to the atomic protocols, for simplicity. Allowing the LLC to directly identify this situation is simpler than requiring the use of NoData messages; with a non-atomic protocol, there can be a large number of NoData messages in the system and NoData messages can arrive before their associated PutM requests.

7.5.5 An Optimized, Non-Stalling MSI Protocol with a Split-Transaction Bus

As mentioned in the previous section, we sacrificed some performance by stalling on the newly possible transitions of the system with the split-transaction bus. For example, a cache with a block in state IS^D stalled instead of processing an Other-GetM for that block. However, it is possible that there are one or more requests after the Other-GetM, to other blocks, that the cache could process without stalling. By stalling a request, the protocol stalls all requests after the stalled request and delays those transactions from completing. Ideally, we would like a coherence controller to process requests behind a request that is stalled, but recall that—to support a total order of memory requests—snooping requires coherence controllers to observe and process requests in the order received. Reordering is not allowed.

The solution to this problem is to process all messages, in order, instead of stalling. Our approach is to add transient states that reflect messages that the coherence controller has received but must remember to complete at a later event. Returning to the example of a cache block in IS^D, if the cache controller observes an Other-GetM on the bus, then it changes the block state to IS^DI (which denotes "in I, going to S, waiting for data, and when data arrives will go to I"). Similarly, a block in IM^D that receives an Other-GetS changes state to IM^DS and must remember the requestor of the Other-GetS. When the data arrive in response to the cache's GetM, the cache controller sends the data to the requestor of the Other-GetS and changes the block's state to S.

In addition to the proliferation of transient states, a non-stalling protocol introduces a potential livelock problem. Consider a cache with a block in IM^DS that receives the data in response to its GetM. If the cache *immediately* changes the block state to S and sends the data to the requestor of the Other-GetS, it does not get to perform the store for which it originally issued its GetM. If the core then re-issues the GetM, the same situation could arise again and again, and the store might never perform. To guarantee that this livelock cannot arise, we require that a cache in IS^DI, IM^DI, IM^DS, or IM^DSI (or any comparable state in a protocol with additional stable coherence states) perform one load or store to the block when it receives the data for its request.[6] After performing one load or store, it may then change state and forward the block to another cache. We defer a more in-depth treatment of livelock to Section 9.3.2.

We present the detailed specification of the non-stalling MSI protocol in Tables 7.20 and 7.21. The most obvious difference is the number of transient states. There is nothing inherently complicated about any of these states, but they do add to the overall complexity of the protocol.

We have not removed the stalls from the memory controller because it is not feasible. Consider a block in $IorS^D$. The memory controller observes a GetM from core C1 and currently stalls.

[6]The load or store must be performed *if and only if* that load or store was the oldest load or store in program order when the coherence request was first issued. We discuss this issue in more detail in Section 9.3.2.

TABLE 7.20: Optimized MSI Snooping with Split-Transaction Bus—Cache Controller

	load	store	replacement	OwnGetS	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own Data response
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-	
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-	**-/IS^{A}**
IS^{D}	stall	stall	stall				**-**	**-/IS^{D}I**		load hit/S
IS^{A}	**stall**	**stall**	**stall**	**load hit/S**			**-**	**-**		
$IS^{D}I$	stall	stall	stall				**-**	**-**		load hit/I
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-	**-/IM^{A}**
IM^{D}	stall	stall	stall				**-/IM^{D}S**	**-/IM^{D}I**		store hit//M
IM^{A}	**stall**	**stall**	**stall**		**store hit/M**		**-**	**-**	-	
$IM^{D}I$	**stall**	**stall**	**stall**				**-**	**-**		**store hit, send data to GetM requestor/I**
$IM^{D}S$	**stall**	**stall**	**stall**				**-**	**-/IM^{D}SI**		**store hit, send data to GetS requestor and mem/S**
$IM^{D}SI$	**stall**	**stall**	**stall**					**-**		**store hit, send data to GetS requestor and mem/I**
S	hit	issue GetM/SM^{AD}	-/I				-	-/I		
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}		**-/SM^{A}**
SM^{D}	hit	stall	stall				**-/SM^{D}S**	**-/SM^{D}I**		store hit/M
SM^{A}	**hit**	**stall**	**stall**		**store hit/M**		**-**	**-/IM^{A}**		
$SM^{D}I$	**hit**	**stall**	**stall**				**-**	**-**		**store hit, send data to GetM requestor/I**
$SM^{D}S$	**hit**	**stall**	**stall**				**-**	**-/SM^{D}SI**		**store hit, send data to GetS requestor and mem/S**
$SM^{D}SI$	**hit**	**stall**	**stall**				**-**	**-**		**store hit, send data to GetS requestor and mem/I**
M	hit	hit	issue PutM/MI^{A}				send data to requestor and to memory/S	send data to requestor/I		
MI^{A}	hit	hit	stall			send data to requestor/I	send data to requestor and to memory/II^{A}	send data to requestor/II^{A}		
II^{A}	stall	stall	stall			-/I	-	-	-	

TABLE 7.21: Optimized MSI Snooping with Split-Transaction Bus—Memory Controller

	GetS	GetM	PutM from Owner	PutM from Non-Owner	Data
IorS	send data to requestor	send data to requestor, set owner to requestor/M		-	
M	clear Owner/IorS^{D}	set Owner to requestor	clear Owner/IorS^{D}	-	**write data to memory/IorS^{A}**
IorS^{D}	**stall**	**stall**	**stall**	-	write data to memory/IorS
IorS^{A}	**clear Owner/IorS**	-	**clear Owner/IorS**	-	

However, it would appear that we could simply change the block's state to IorS^{D}M while waiting for the data. Yet, while in IorS^{D}M, the memory controller could observe a GetS from core C2. If the memory controller does not stall on this GetS, it must change the block state to IorS^{D}MIorS^{D}. In this state, the memory controller could observe a GetM from core C3. There is no elegant way to bound the number of transient states needed at the LLC/memory to a small number (i.e., smaller than the number of cores) and so, for simplicity, we have the memory controller stall.

7.6 OPTIMIZATIONS TO THE BUS INTERCONNECTION NETWORK

So far in this chapter we have assumed system models in which there exists a single shared-wire bus for coherence requests and responses or dedicated shared-wire buses for requests and responses. In this section, we explore two other possible system models that enable improved performance.

7.6.1 Separate Non-Bus Network for Data Responses

We have emphasized the need of snooping systems to provide a total order of broadcast coherence requests. The example in Table 7.2 showed how the lack of a total order of coherence requests can lead to incoherence. However, there is no such need to order coherence *responses*, nor is there a need to broadcast them. Thus, coherence responses could travel on a separate network that does not support broadcast or ordering. Such networks include crossbars, meshes, tori, butterflies, etc.

There are several advantages to using a separate, non-bus network for coherence responses.

- Implementability: it is difficult to implement high-speed shared-wire buses, particularly for systems with many controllers on the bus. Other topologies can use point-to-point links.
- Throughput: a bus can provide only one response at a time. Other topologies can have multiple responses in-flight at a time.
- Latency: using a bus for coherence responses requires that each response incur the latency to arbitrate for the bus. Other topologies can allow responses to be sent immediately without arbitration.

7.6.2 Logical Bus for Coherence Requests

Snooping systems require that there exist a total order of broadcast coherence requests. A shared-wire bus for coherence requests is the most straightforward way to achieve this total order of broadcasts, but it is not the only way to do so. There are two ways to achieve the same totally ordered broadcast properties as a bus (i.e., a logical bus) without having a physical bus.

- Other topologies with physical total order: a shared-wire bus is the most obvious topology for achieving a total order of broadcasts, but other topologies exist. One notable example is a tree with the coherence controllers at the leaves of the tree. If all coherence requests are unicasted to the root of the tree and then broadcast down the tree, then each coherence controller observes the same total order of coherence broadcasts. The serialization point in this topology is the root of the tree. Sun Microsystems used a tree topology in its Starfire multiprocessor [3], which we discuss in detail in Section 7.7.
- Logical total order: a total order of broadcasts can be obtained even without a network topology that naturally provides such an order. The key is to order the requests in logical time. Martin et al. [6] designed a snooping protocol, called Timestamp Snooping, that can function on any network topology. To issue a coherence request, a cache controller broadcasts it to every coherence controller and labels the broadcast with the logical time at which the broadcast message should be ordered. The protocol must ensure that (a) every broadcast has a distinct logical time, (b) coherence controllers process requests in logical time order (even when they arrive out of this order in physical time), and (c) no request at logical time T can arrive at a controller after that controller has passed logical time T. Agarwal et al. recently proposed a similar scheme called In-Network Snoop Ordering (INSO) [1].

Flashback to Quiz Question 7: A snooping cache coherence protocol requires the cores to communicate on a bus. *True or false?*
Answer: *False!* Snooping requires a totally ordered broadcast network, but that functionality can be implemented without a physical bus.

7.7 CASE STUDIES

We present two examples of real-world snooping systems: the Sun Starfire E10000 and the IBM Power5.

7.7.1 Sun Starfire E10000

Sun Microsystems's Starfire E10000 [3] is an interesting example of a commercial system with a snooping protocol. The coherence protocol itself is not that remarkable; the protocol is a typical

MOESI snooping protocol with write-back caches. What distinguishes the E10000 is how it was designed to scale up to 64 processors. The architects innovated based on three important observations, which we discuss in turn.

First, shared-wire snooping buses do not scale to large numbers of cores, largely due to electrical engineering constraints. In response to this observation, the E10000 uses only point-to-point links instead of buses. Instead of broadcasting coherence requests on physical (shared-wire) buses, the E10000 broadcasts coherence requests on a *logical* bus. The key insight behind snooping protocols is that they require a total order of coherence requests, but this total order does not require a physical bus. As illustrated in Figure 7.12, the E10000 implements a logical bus as a tree, in which the processors are the leaves. All links in the tree are point-to-point, thus eliminating the need for buses. A processor unicasts a request up to the top of the tree, where it is serialized and then broadcast down the tree. Because of the serialization at the root, the tree provides totally ordered broadcast. A given request may arrive at two processors at different times, which is fine; the important constraint is that the processors observe the same total order of requests.

The second observation made by the E10000 architects is that greater coherence request bandwidth can be achieved by using multiple (logical) buses, while still maintaining a total order of coherence requests. The E10000 has four logical buses, and coherence requests are address-interleaved across them. A total order is enforced by requiring processors to snoop the logical buses in a fixed, pre-determined order.

Third, the architects observed that data response messages, which are much larger than request messages, do not require the totally ordered broadcast network required for coherence requests.

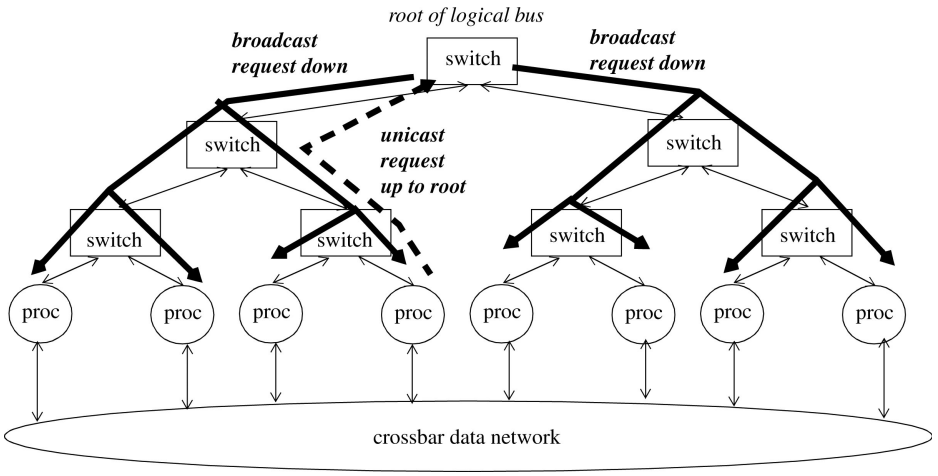

FIGURE 7.12: Starfire E10000 (drawn with only eight processors for clarity). A coherence request is unicast up to the root, where it is serialized, before being broadcast down to all processors.

Many prior snooping systems implemented a data bus, which needlessly provides both broadcasting and total ordering, while limiting bandwidth. To improve bandwidth, the E10000 implements the data network as a crossbar. Once again, there are point-to-point links instead of buses, and the bandwidth of the crossbar far exceeds what would be possible with a bus (physical or logical).

The architecture of the E10000 has been optimized for scalability, and this optimized design requires the architects to reason about non-atomic requests and non-atomic transactions.

7.7.2 IBM Power5

The IBM Power5 [8] is a 2-core chip in which both cores share an L2 cache. Each Power5 chip has a fabric bus controller (FBC) that enables multiple Power5 chips to be connected together to create larger systems. Large systems contain up to eight nodes, where each node is a multi-chip module (MCM) with four Power5 chips.

Viewed abstractly, the IBM Power5 appears to use a fairly typical MESI snooping protocol implemented atop a split-transaction bus. However, this simplistic description misses several unique features that are worth discussing. In particular, we focus on two aspects: the ring topology of the interconnection network and the addition of novel variants of the MESI coherence states.

7.7.2.1 Snooping Coherence on a Ring

The Power5 uses an interconnection network that is quite different from what we have discussed thus far, and these differences have important impacts on the coherence protocol. Most significantly, the Power5 connects nodes with three unidirectional rings, which are used for carrying three types of messages: requests, snoop responses/decision messages, and data. Unidirectional rings do not provide a total order, unless all messages are required to start from the same node on the ring, which the Power5 does not. Rather, the requestor sends a request message around the ring and then absorbs the request when it sees it arrive back after traveling the entire ring. Each node observes the request on the ring and every processor in the node determines its snoop response. The first node to observe the request provides a single snoop response that is the aggregated snoop response of all of the processors on that node. A snoop response is *not* an actual data response, but rather a description of the action the chip or node would take. Without a totally ordered network, the chips/nodes cannot immediately act because they might not make consistent decisions about how to respond. The snoop response travels on the snoop response ring to the next node. This node similarly produces a single snoop response that aggregates the snoop response of the first node plus the snoop responses of all processors on the second node. When the aggregated snoop response of all nodes reaches the requestor chip, the requestor chip determines what every processor should do to respond to the request. The requestor chip broadcasts this decision along the ring to every node. This decision

message is processed by every node/chip in the ring, and the node/chip that has been determined to be the one to provide a data response sends that data response on the data ring to the requestor.

This protocol is far more complicated than typical snooping protocols because of the lack of a totally ordered interconnection network. The protocol still has a total logical order of coherence requests, but without a totally ordered network, a node cannot immediately respond to a request because the request's position in the total order has not yet been determined by when it appears on the network. Despite the complexity, the Power5 design offers the advantages of having only point-to-point links and the simplicity of the ring topology (e.g., routing in a ring is so simple that switching can be faster than for other topologies). There have been other protocols that have exploited ring topologies and explored ordering issues for rings [2, 4, 7].

7.7.2.2 Extra Variants of Coherence States

The Power5 protocol is fundamentally a MESI protocol, but it has several "flavors" of some of these states. We list all of the states in Table 7.22. There are two new states that we wish to highlight. First, there is the SL variant of the Shared state. If an L2 cache holds a block in state SL, it may respond with data to a GetS from a processor on the same node, thus reducing this transaction's latency and reducing off-chip bandwidth demand; this ability to provide data distinguishes SL from S.

TABLE 7.22: Power5 L2 Cache Coherence States

State	Pemissions	Description
I	none	Invalid
S	read-only	Shared
SL	**read-only**	**Shared local data source, but can respond with data to requests from processors in same node (sometimes referred to as F state, as in Intel QuickPath protocol (Section 8.8.4)**
S (S)	read-only	Shared
Me (E)	read-write	Exclusive
M (M)	read-write	Modified
Mu	**read-write**	**Modified unsolicited - received read-write data in response to read-only request**
T	**read-only**	**Tagged - was M, received GetS. T is sometime described as being a read-write state, which violates the SWMR invariant since there are also blocks in state S. A better way to think of T is that it is like E: it can immediately transition to M. However, unlike E, this transition is not silent: a store to a block in T state immediately transitions to M but (atomically) issues an invalidation message on the ring. Although other caches may race with this request, the T state has priority, and thus is guaranteed to be ordered first and thus does not need to wait for the invalidations to complete.**

The other interesting new state is the T(agged) state. A block enters the T state when it is in Modified and receives a GetS request. Instead of downgrading to S, which it would do in a MESI protocol, or O, which it would do in a MOSI protocol, the cache changes the state to T. A block in state T is similar to the O state, in that it has a value that is more recent than the value in memory, and there may or may not be copies of the block in state S in other caches. Like the O state, the block may be read in the T state. Surprisingly, the T state is sometimes described as being a read-write state, which violates the SWMR invariant. Indeed, a store to state T may be performed immediately, and thus indeed violates the SWMR invariant in real (physical) time. However, the protocol still enforces the SWMR invariant in a logical time based on ring-order. Although the details of this ordering are beyond the scope of this primer, we think it helps to think of the T state as a variation on the E state. Recall that the E state allows a silent transition to M; thus a store to a block in state E may be immediately performed, so long as the state (atomically) transitions to state M. The T state is similar; a store in state T immediately transitions to state M. However, because there may also be copies in state S, a store in state T also causes the immediate issue of an invalidation message on the ring. Other cores may be attempting to upgrade from I or S to M, but the T state acts as the coherence ordering point and thus has priority and need not wait for an acknowledgment. It is not clear that this protocol is sufficient to support strong memory consistency models such as SC and TSO; however, as we discussed in Chapter 5, the Power memory model is one of the weakest memory consistency models. This Tagged state optimizes the common scenario of producer-consumer sharing, in which one thread writes a block and one or more other threads then read that block. The producer can re-obtain read-write access without having to wait as long each time.

7.8 DISCUSSION AND THE FUTURE OF SNOOPING

Snooping systems were prevalent in early multiprocessors because of their reputed simplicity and because their lack of scalability did not matter for the relatively small systems that dominated the market. Snooping also offers performance advantages for non-scalable systems because every snooping transaction can be completed with two messages, which we will contrast against the three-message transactions of directory protocols.

Despite its advantages, snooping is no longer commonly used. Even for small-scale systems, where snooping's lack of scalability is not a concern, snooping is no longer common. Snooping's requirement of a totally ordered broadcast network is just too costly, compared to the low-cost interconnection networks that suffice for directory protocols. Furthermore, for scalable systems, snooping is clearly a poor fit. Systems with very large numbers of cores are likely to be bottlenecked by both the interconnection network bandwidth needed to broadcast requests and the coherence controller bandwidth required to snoop every request. For such systems, a more scalable coherence

protocol is required, and it is this need for scalability that originally motivated the directory protocols we present in the next chapter.

7.9 REFERENCES

[1] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects. In *Proceedings of the Fourteenth International Symposium on High-Performance Computer Architecture*, pp. 67–78, Feb. 2009. doi:10.1109/HPCA.2009.4798238

[2] L. A. Barroso and M. Dubois. Cache Coherence on a Slotted Ring. In *Proceedings of the 20th International Conference on Parallel Processing*, Aug. 1991.

[3] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998. doi:10.1109/40.653032

[4] S. Frank, H. Burkhardt, III, and J. Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of the 38th Annual IEEE Computer Society Computer Conference (COMPCON)*, pp. 285–95, Feb. 1993.

[5] M. Galles and E. Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In *Proceedings of the Hawaii International Conference on System Sciences*, 1994. doi:10.1109/HICSS.1994.323177

[6] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 25–36, Nov. 2000.

[7] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006. doi:10.1109/MICRO.2006.14

[8] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5), July/September 2005. doi:10.1147/rd.494.0505

CHAPTER 8

Directory Coherence Protocols

In this chapter, we present directory coherence protocols. Directory protocols were originally developed to address the lack of scalability of snooping protocols. Traditional snooping systems broadcast all requests on a totally ordered interconnection network and all requests are snooped by all coherence controllers. By contrast, directory protocols use a level of indirection to avoid both the ordered broadcast network and having each cache controller process every request.

We first introduce directory protocols at a high level (Section 8.1). We then present a system with a complete but unsophisticated three-state (MSI) directory protocol (Section 8.2). This system and protocol serve as a baseline upon which we later add system features and protocol optimizations. We then explain how to add the Exclusive state (Section 8.3) and the Owned state (Section 8.4) to the baseline MSI protocol. Next we discuss how to represent the directory state (Section 8.5) and how to design and implement the directory itself (Section 8.6). We then describe techniques for improving performance and reducing the implementation costs (Section 8.7). We then discuss commercial systems with directory protocols (Section 8.8) before concluding the chapter with a discussion of directory protocols and their future (Section 8.9).

Those readers who are content to learn just the basics of directory coherence protocols can skim or skip Section 8.3 through Section 8.7, although some of the material in these sections will help the reader to better understand the case studies in Section 8.8.

8.1 INTRODUCTION TO DIRECTORY PROTOCOLS

The key innovation of directory protocols is to establish a *directory* that maintains a global view of the coherence state of each block. The directory tracks which caches hold each block and in what states. A cache controller that wants to issue a coherence request (e.g., a GetS) sends it directly to the directory (i.e., a unicast message), and the directory looks up the state of the block to determine what actions to take next. For example, the directory state might indicate that the requested block is owned by core C2's cache and thus the request should be forwarded to C2 (e.g., using a new Fwd-GetS request) to obtain a copy of the block. When C2's cache controller receives this forwarded request, it unicasts a response to the requesting cache controller.

It is instructive to compare the basic operation of directory protocols and snooping protocols. In a directory protocol, the directory maintains the state of each block, and cache controllers send all requests to the directory. The directory either responds to the request or forwards the request to one or more other coherence controllers that then respond. Coherence transactions typically involve either two steps (a unicast request, followed by a unicast response) or three steps (a unicast request, $K \geq 1$ forwarded requests, and K responses, where K is the number of sharers). Some protocols even have a fourth step, either because responses indirect through the directory or because the requestor notifies the directory on transaction completion. In contrast, snooping protocols distribute a block's state across potentially all of the coherence controllers. Because there is no central summary of this distributed state, coherence requests must be broadcast to all coherence controllers. Snooping coherence transactions thus always involve two steps (a broadcast request, followed by a unicast response).

Like snooping protocols, a directory protocol needs to define when and how coherence transactions become ordered with respect to other transactions. In most directory protocols, a coherence transaction is ordered at the directory. Multiple coherence controllers may send coherence requests to the directory at the same time, and the transaction order is determined by the order in which the requests are serialized at the directory. If two requests race to the directory, the interconnection network effectively chooses which request the directory will process first. The fate of the request that arrives second is a function of the directory protocol and what types of requests are racing. The second request might get (a) processed immediately after the first request, (b) held at the directory while awaiting the first request to complete, or (c) negatively acknowledged (NACKed). In the latter case, the directory sends a negative acknowledgment message (NACK) to the requestor, and the requestor must re-issue its request. In this chapter, we do not consider protocols that use NACKs, but we do discuss the possible use of NACKs and how they can cause livelock problems in Section 9.3.2.

Using the directory as the ordering point represents another key difference between directory protocols and snooping protocols. Traditional snooping protocols create a total order by serializing all transactions on the ordered broadcast network. Snooping's total order not only ensures that each block's requests are processed in per-block order but also facilitates implementing a memory consistency model. Recall that traditional snooping protocols use totally ordered broadcast to serialize all requests; thus, when a requestor observes its own coherence request this serves as notification that its coherence epoch may begin. In particular, when a snooping controller sees its own GetM request, it can infer that other caches will invalidate their S blocks. We demonstrated in Table 7.4 that this serialization notification is sufficient to support the strong SC and TSO memory consistency models.

In contrast, a directory protocol orders transactions at the directory to ensure that conflicting requests are processed by all nodes in per-block order. However, the lack of a total order means that a requestor in a directory protocol needs another strategy to determine when its request has been serialized and thus when its coherence epoch may safely begin. Because (most) directory protocols do not use totally ordered broadcast, there is no global notion of serialization. Rather, a request must be individually serialized with respect to all the caches that (may) have a copy of the block. Explicit messages are needed to notify the requestor that its request has been serialized by each relevant cache. In particular, on a GetM request, each cache controller with a shared (S) copy must send an explicit acknowledgment (Ack) message once it has serialized the invalidation message.

This comparison between directory and snooping protocols highlights the fundamental trade-off between them. A directory protocol achieves greater scalability (i.e., because it requires less bandwidth) at the cost of a level of indirection (i.e., having three steps, instead of two steps, for some transactions). This additional level of indirection increases the latency of some coherence transactions.

8.2 BASELINE DIRECTORY SYSTEM

In this section, we present a baseline system with a straightforward, modestly optimized directory protocol. This system provides insight into the key features of directory protocols while revealing inefficiencies that motivate the features and optimizations presented in subsequent sections of this chapter.

8.2.1 Directory System Model

We illustrate our directory system model in Figure 8.1. Unlike for snooping protocols, the topology of the interconnection network is intentionally vague. It could be a mesh, torus, or any other topology that the architect wishes to use. One restriction on the interconnection network that we assume in this chapter is that it enforces point-to-point ordering. That is, if controller A sends two messages to controller B, then the messages arrive at controller B in the same order in which they were sent.[1] Having point-to-point ordering reduces the complexity of the protocol, and we defer a discussion of networks without ordering until Section 8.7.3.

The only differences between this directory system model and the baseline system model in Figure 2.1 is that we have added a directory and we have renamed the memory controller to be the

[1] Strictly speaking, we require point-to-point order for only certain types of messages, but this is a detail that we defer until Section 8.7.3.

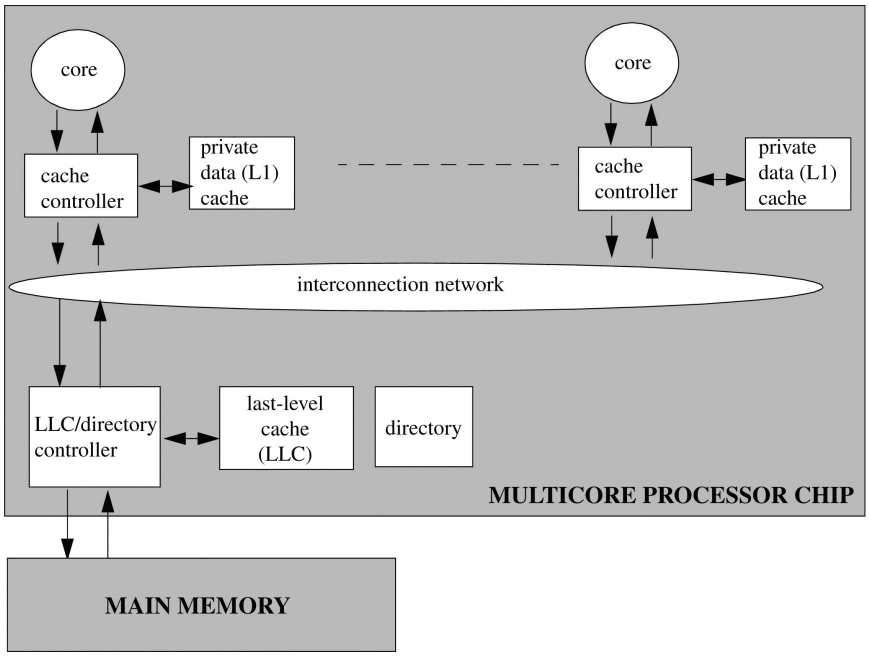

FIGURE 8.1: Directory system model.

directory controller. There are many ways of sizing and organizing the directory, and for now we assume the simplest model: for each block in memory, there is a corresponding directory entry. In Section 8.6, we examine and compare more practical directory organization options. We also assume a monolithic LLC with a single directory controller; in Section 8.7.1, we explain how to distribute this functionality across multiple banks of an LLC and multiple directory controllers.

8.2.2 High-Level Protocol Specification

The baseline directory protocol has only three stable states: MSI. A block is owned by the directory controller unless the block is in a cache in state M. The directory state for each block includes the stable coherence state, the identity of the owner (if the block is in state M), and the identities of the

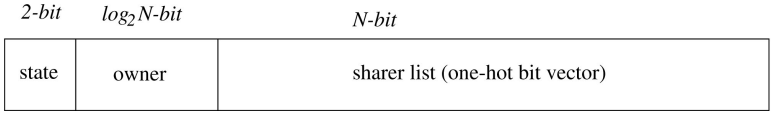

FIGURE 8.2: Directory entry for a block in a system with N nodes.

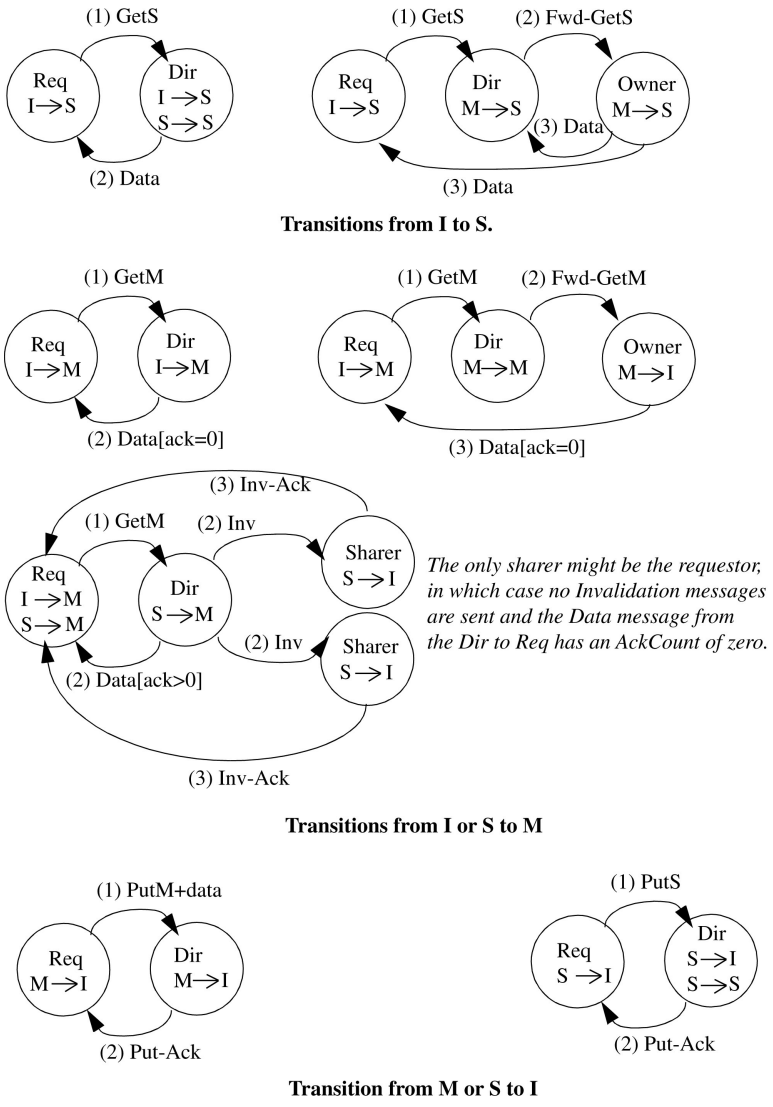

FIGURE 8.3: High-Level Description of MSI Directory Protocol. In each transition, the cache controller that requests the transaction is denoted "Req".

sharers encoded as a one-hot bit vector (if the block is in state S). We illustrate a directory entry in Figure 8.2. In Section 8.5, we will discuss other encodings of directory entries.

Before presenting the detailed specification, we first illustrate a higher level abstraction of the protocol in order to understand its fundamental behaviors. In Figure 8.3, we show the transactions in which a cache controller issues coherence requests to change permissions from I to S, I or S to

M, M to I, and S to I. As with the snooping protocols in the last chapter, we specify the directory state of a block using a cache-centric notation (e.g., a directory state of M denotes that there exists a cache with the block in state M). Note that a cache controller may not silently evict a Shared block; that is, there is an explicit PutS request. We defer a discussion of protocols with silent evictions of shared blocks, as well as a comparison of silent versus explicit PutS requests, until Section 8.7.4.

Most of the transactions are fairly straightforward, but two transactions merit further discussion here. The first is the transaction that occurs when a cache is trying to upgrade permissions from I or S to M and the directory state is S. The cache controller sends a GetM to the directory, and the directory takes two actions. First, it responds to the requestor with a message that includes the data and the "AckCount"; the AckCount is the number of current sharers of the block. The directory sends the AckCount to the requestor to inform the requestor of how many sharers must acknowledge having invalidated their block in response to the GetM. Second, the directory sends an Invalidation (Inv) message to all of the current sharers. Each sharer, upon receiving the Invalidation, sends an Invalidation-Ack (Inv-Ack) to the requestor. Once the requestor receives the message from the directory and *all* of the Inv-Ack messages, it completes the transaction. The requestor, having received all of the Inv-Ack messages, knows that there are no longer any readers of the block and thus it may write to the block without violating coherence.

The second transaction that merits further discussion occurs when a cache is trying to evict a block in state M. In this protocol, we have the cache controller send a PutM message that includes the data to the directory. The directory responds with a Put-Ack. If the PutM did not carry the data with it, then the protocol would require a third message—a data message from the cache controller to the directory with the evicted block that had been in state M—to be sent in a PutM transaction. The PutM transaction in this directory protocol differs from what occurred in the snooping protocol, in which a PutM did not carry data.

8.2.3 Avoiding Deadlock

In this protocol, the reception of a message can cause a coherence controller to send another message. In general, if event A (e.g., message reception) can cause event B (e.g., message sending) and both these events require resource allocation (e.g., network links and buffers), then we must be careful to avoid deadlock that could occur if circular resource dependences arise. For example, a GetS request can cause the directory controller to issue a Fwd-GetS message; if these messages use the same resources (e.g., network links and buffers), then the system can potentially deadlock. In Figure 8.4, we illustrate a deadlock in which two coherence controllers C1 and C2 are responding to each other's requests, but the incoming queues are already full of other coherence requests. If the queues are FIFO, then the responses cannot pass the requests. Because the queues are full, each con-

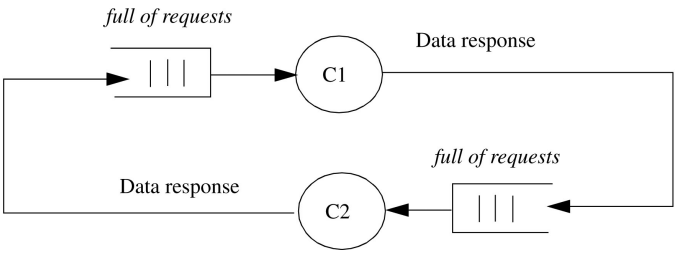

FIGURE 8.4: Deadlock example.

troller stalls trying to send a response. Because the queues are FIFO, the controller cannot switch to work on a subsequent request (or get to the response). Thus, the system deadlocks.

A well-known solution for avoiding deadlock in coherence protocols is to use separate networks for each class of message. The networks can be physically separate or logically separate (called *virtual networks*), but the key is avoiding dependences between classes of messages. Figure 8.5 illustrates a system in which request and response messages travel on separate physical networks. Because a response cannot be blocked by another request, it will eventually be consumed by its destination node, breaking the cyclic dependence.

The directory protocol in this section uses three networks to avoid deadlock. Because a request can cause a forwarded request and a forwarded request can cause a response, there are three message classes that each require their own network. Request messages are GetS, GetM, and PutM. Forwarded request messages are Fwd-GetS, Fwd-GetM, Inv(alidation), and Put-Ack. Response messages are Data and Inv-Ack. The protocols in this chapter require that the Forwarded Request network provides point-to-point ordering; other networks have no ordering constraints nor are there any ordering constraints between messages traveling on different networks.

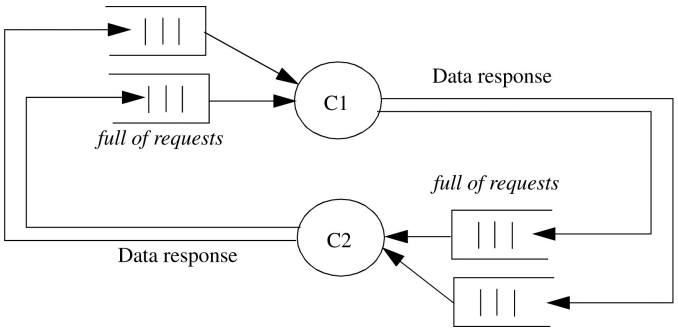

FIGURE 8.5: Avoiding deadlock with separate networks.

We defer a more thorough discussion of deadlock avoidance, including more explanation of virtual networks and the exact requirements for avoiding deadlock, until Section 9.3.

8.2.4 Detailed Protocol Specification

We present the detailed protocol specification, including all transient states, in Tables 8.1 and 8.2. Compared to the high-level description in Section 8.2.2, the most significant difference is the transient states. The coherence controllers must manage the states of blocks that are in the midst of coherence transactions, including situations in which a cache controller receives a forwarded request from another controller in between sending its coherence request to the directory and receiving all of its necessary response messages, including Data and possible Inv-Acks. The cache controllers can maintain this state in the miss status handling registers (MSHRs) that cores use to keep track of outstanding coherence requests. Notationally, we represent these transient states in the form XY^{AD}, where the superscript A denotes waiting for acknowledgments and the superscript D denotes waiting for data. (This notation differs from the snooping protocols, in which the superscript A denoted waiting for a request to appear on the bus.)

TABLE 8.1: MSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			stall		-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall						ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall						ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

TABLE 8.2: MSI Directory Protocol—Directory Controller

	GetS	GetM	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data
I	send data to Req, add Req to Sharers/S	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req	
S	send data to Req, add Req to Sharers	send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I		remove Req from Sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/S^D	Send Fwd-GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to memory, clear Owner, send Put-Ack to Req/I	send Put-Ack to Req	
S^D	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req		remove Req from Sharers, send Put-Ack to Req	copy data to memory/S

Because these tables can be somewhat daunting at first glance, the next section walks through some example scenarios.

8.2.5 Protocol Operation

The protocol enables caches to acquire blocks in states S and M and to replace blocks to the directory in either of these states.

I to S (common case #1)

The cache controller sends a GetS request to the directory and changes the block state from I to IS^D. The directory receives this request and, if the directory is the owner (i.e., no cache currently has the block in M), the directory responds with a Data message, changes the block's state to S (if it is not S already), and adds the requestor to the sharer list. When the Data arrives at the requestor, the cache controller changes the block's state to S, completing the transaction.

I to S (common case #2)

The cache controller sends a GetS request to the directory and changes the block state from I to IS^D. If the directory is *not* the owner (i.e., there is a cache that currently has the block in M), the directory forwards the request to the owner and changes the block's state to the transient state S^D. The owner responds to this Fwd-GetS message by sending Data to the requestor and changing the block's state to S. The now-previous owner must also send Data to the directory since it is relinquishing ownership to the directory, which must have an up-to-date copy of the block. When the

Data arrives at the requestor, the cache controller changes the block state to S and considers the transaction complete. When the Data arrives at the directory, the directory copies it to memory, changes the block state to S, and considers the transaction complete.

I to S (race cases)

The above two I-to-S scenarios represent the common cases, in which there is only one transaction for the block in progress. Most of the protocol's complexity derives from having to deal with the less-common cases of multiple in-progress transactions for a block. For example, a reader may find it surprising that a cache controller can receive an Invalidation for a block in state IS^D. Consider core C1 that issues a GetS and goes to IS^D and another core C2 that issues a GetM for the same block that arrives at the directory after C1's GetS. The directory first sends C1 Data in response to its GetS and then an Invalidation in response to C2's GetM. Because the Data and Invalidation travel on separate networks, they can arrive out of order, and thus C1 can receive the Invalidation before the Data.

I or S to M

The cache controller sends a GetM request to the directory and changes the block's state from I to IM^{AD}. In this state, the cache waits for Data and (possibly) Inv-Acks that indicate that other caches have invalidated their copies of the block in state S. The cache controller knows how many Inv-Acks to expect, since the Data message contains the AckCount, which may be zero. Figure 8.3 illustrates the three common-case scenarios of the directory responding to the GetM request. If the directory is in state I, it simply sends Data with an AckCount of zero and goes to state M. If in state M, the directory controller forwards the request to the owner and updates the block's owner; the now-previous owner responds to the Fwd-GetM request by sending Data with an AckCount of zero. The last common case occurs when the directory is in state S. The directory responds with Data and an AckCount equal to the number of sharers, plus it sends Invalidations to each core in the sharer list. Cache controllers that receive Invalidation messages invalidate their shared copies and send Inv-Acks to the requestor. When the requestor receives the last Inv-Ack, it transitions to state M. Note the special Last-Inv-Ack event in Table 8.1, which simplifies the protocol specification.

These common cases neglect some possible races that highlight the concurrency of directory protocols. For example, core C1 has the cache block in state IM^A and receives a Fwd-GetS from C2's cache controller. This situation is possible because the directory has already sent Data to C1, sent Invalidation messages to the sharers, and changed its state to M. When C2's GetS arrives at the directory, the directory simply forwards it to the owner, C1. This Fwd-GetS may arrive at C1 before all of the Inv-Acks arrive at C1. In this situation, our protocol simply stalls and the cache controller

waits for the Inv-Acks. Because Inv-Acks travel on a separate network, they are guaranteed not to block behind the unprocessed Fwd-GetS.

M to I

To evict a block in state M, the cache controller sends a PutM request that includes the data and changes the block state to MI^A. When the directory receives this PutM, it updates the LLC/memory, responds with a Put-Ack, and transitions to state I. Until the requestor receives the Put-Ack, the block's state remains effectively M and the cache controller must respond to forwarded coherence requests for the block. In the case where the cache controller receives a forwarded coherence request (Fwd-GetS or Fwd-GetM) between sending the PutM and receiving the Put-Ack, the cache controller responds to the Fwd-GetS or Fwd-GetM and changes its block state to SI^A or II^A, respectively. These transient states are effectively S and I, respectively, but denote that the cache controller must wait for a Put-Ack to complete the transition to I.

S to I

Unlike the snooping protocols in the previous chapter, our directory protocols do not silently evict blocks in state S. Instead, to replace a block in state S, the cache controller sends a PutS request and changes the block state to SI^A. The directory receives this PutS and responds with a Put-Ack. Until the requestor receives the Put-Ack, the block's state is effectively S. If the cache controller receives an Invalidation request after sending the PutS and before receiving the Put-Ack, it changes the block's state to II^A. This transient state is effectively I, but it denotes that the cache controller must wait for a Put-Ack to complete the transaction from S to I.

8.2.6 Protocol Simplifications

This protocol is relatively straightforward and sacrifices some performance to achieve this simplicity. We now discuss two simplifications:

- The most significant simplification, other than having only three stable states, is that the protocol stalls in certain situations. For example, a cache controller stalls when it receives a forwarded request while in a transient state. A higher performance option, discussed in Section 8.7.2, would be to process the messages and add more transient states.
- A second simplification is that the directory sends Data (and the AckCount) in response to a cache that is changing a block's state from S to M. The cache already has valid data and thus it would be sufficient for the directory to simply send a data-less AckCount. We defer adding this new type of message until we present the MOSI protocol in Section 8.4.

8.3 ADDING THE EXCLUSIVE STATE

As we previously discussed in the context of snooping protocols, adding the Exclusive (E) state is an important optimization because it enables a core to read and then write a block with only a single coherence transaction, instead of the two required by an MSI protocol. At the highest level, this optimization is independent of whether the cache coherence uses snooping or directories. If a core issues a GetS and the block is not currently shared by other cores, then the requestor may obtain the block in state E. The core may then silently upgrade the block's state from E to M without issuing another coherence request.

In this section, we add the E state to our baseline MSI directory protocol. As with the MESI snooping protocol in the previous chapter, the operation of the protocol depends on whether the E state is considered an ownership state or not. And, as with the MESI snooping protocol, the primary operational difference involves determining which coherence controller should respond to a request for a block that the directory gave to a cache in state E. The block may have been silently upgraded from E to M since the directory gave the block to the cache in state E.

In protocols in which an E block is owned, the solution is simple. The cache with the block in E (or M) is the owner and thus must respond to requests. A coherence request sent to the directory will be forwarded to the cache with the block in state E. Because the E state is an ownership state, the eviction of an E block cannot be performed silently; the cache must issue a PutE request to the directory. Without an explicit PutE, the directory would not know that the directory was now the owner and should respond to incoming coherence requests. Because we assume in this primer that blocks in E are owned, this simple solution is what we implement in the MESI protocol in this section.

In protocols in which an E block is not owned, an E block can be silently evicted, but the protocol complexity increases. Consider the case where core C1 obtains a block in state E and then the directory receives a GetS or GetM from core C2. The directory knows that C1 is either i) still in state E, ii) in state M (if C1 did a store with a silent upgrade from E to M), or iii) in state I (if the protocol allows C1 to perform a silent PutE). If C1 is in M, the directory must forward the request to C1 so that C1 can supply the latest version of the data. If C1 is in E, C1 or the directory may respond since they both have the same data. If C1 is in I, the directory must respond. One solution, which we describe in more detail in our case study on the SGI Origin [10] in Section 8.8.1, is to have both C1 and the directory respond. Another solution is to have the directory forward the request to C1. If C1 is in I, C1 notifies the directory to respond to C2; else, C1 responds to C2 and notifies the directory that it does not need to respond to C2.

8.3.1 High-Level Protocol Specification

We specify a high-level view of the transactions in Figure 8.6, with differences from the MSI protocol highlighted. There are only two significant differences. First, there is a transition from I to E

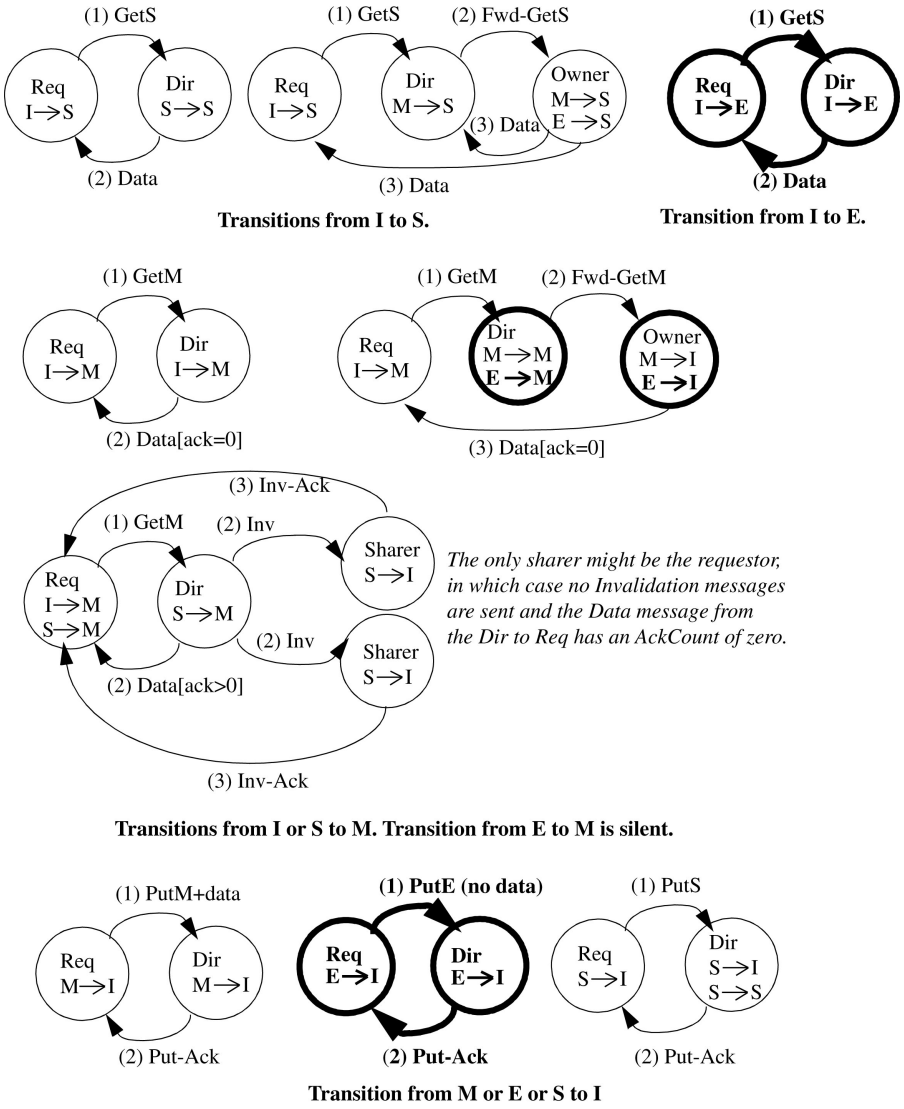

FIGURE 8.6: High-Level Description of MESI Directory Protocol. In each transition, the cache controller that requests the transaction is denoted "Req".

that can occur if the directory receives a GetS for a block in state I. Second, there is a PutE transaction for evicting blocks in state E. Because E is an ownership state, an E block cannot be silently evicted. Unlike a block in state M, the E block is clean, and thus the PutE does not need to carry data; the directory already has the most up-to-date copy of the block.

8.3.2 Detailed Protocol Specification

In Tables 8.3 and 8.4, we present the detailed specification of the MESI protocol, including transient states. Differences with respect to the MSI protocol are highlighted with boldface font. The protocol adds to the set of cache states both the stable E state as well as transient states to handle transactions for blocks initially in state E.

This protocol is somewhat more complex than the MSI protocol, with much of the added complexity at the directory controller. In addition to having more states, the directory controller must distinguish between more possible events. For example, when a PutS arrives, the directory must distinguish whether this is the "last" PutS; that is, did this PutS arrive from the only current sharer? If this PutS is the last PutS, then the directory's state changes to I.

TABLE 8.3: MESI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Exclusive data from Dir	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}											
IS^D	stall	stall	stall			stall		**-/E**	-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall				-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall							ack--	-/M
S	hit	send GetM to Dir/SMAD	send PutS to Dir/SI^A			send Inv-Ack to Req/I							
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}			-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall							ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I								
E	**hit**	**hit/M**	**send PutE (no data) to Dir/EI^A**	**send data to Req and Dir/S**	**send data to Req/I**								
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I						
EI^A	**stall**	**stall**	**stall**	**send data to Req and Dir/SI^A**	**send data to Req/II^A**		**-/I**						
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I						
II^A	stall	stall	stall				-/I						

TABLE 8.4: MESI Directory Protocol—Directory Controller

	GetS	GetM	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM from Non-Owner	PutE (no data) from Owner	PutE from Non-Owner	Data
I	send Exclusive data to Req, set Owner to Req/E	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req		**send Put-Ack to Req**	
S	send data to Req, add Req to Sharers	send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I		remove Req from Sharers, send Put-Ack to Req		**remove Req from Sharers, send Put-Ack to Req**	
E	**forward GetS to Owner, make Owner sharer, add Req to Sharers, clear Owner/S^D**	**forward GetM to Owner, set Owner to Req/M**	**send Put-Ack to Req**	**send Put-Ack to Req**	**copy data to mem, send Put-Ack to Req, clear Owner/I**	**send Put-Ack to Req**	**send Put-Ack to Req, clear Owner/I**	**send Put-Ack to Req**	
M	forward GetS to Owner, make Owner sharer, add Req to Sharers, clear Owner/S^D	forward GetM to owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to mem, send Put-Ack to Req, clear Owner/I	send Put-Ack to Req		**send Put-Ack to Req**	
S^D	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req		remove Req from Sharers, send Put-Ack to Req		**remove Req from Sharers, send Put-Ack to Req**	copy data to LLC/mem/S

8.4 ADDING THE OWNED STATE

For the same reason we added the Owned state to the baseline MSI snooping protocol in Chapter 7, an architect may want to add the Owned state to the baseline MSI directory protocol presented in Section 8.2. Recall from Chapter 2 that if a cache has a block in the Owned state, then the block is valid, read-only, dirty (i.e., it must eventually update memory), and owned (i.e., the cache must respond to coherence requests for the block). Adding the Owned state changes the protocol, compared to MSI, in three important ways: (1) a cache with a block in M that observes a Fwd-GetS changes its state to O and does not need to (immediately) copy the data back to the LLC/memory, (2) more coherence requests are satisfied by caches (in O state) than by the LLC/memory, and (3) there are more 3-hop transactions (which would have been satisfied by the LLC/memory in an MSI protocol).

8.4.1 High-Level Protocol Specification

We specify a high-level view of the transactions in Figure 8.7, with differences from the MSI protocol highlighted. The most interesting difference is the transaction in which a requestor of a block in state I or S sends a GetM to the directory when the block is in the O state in the owner cache and in the S state in one or more sharer caches. In this case, the directory forwards the GetM to the owner,

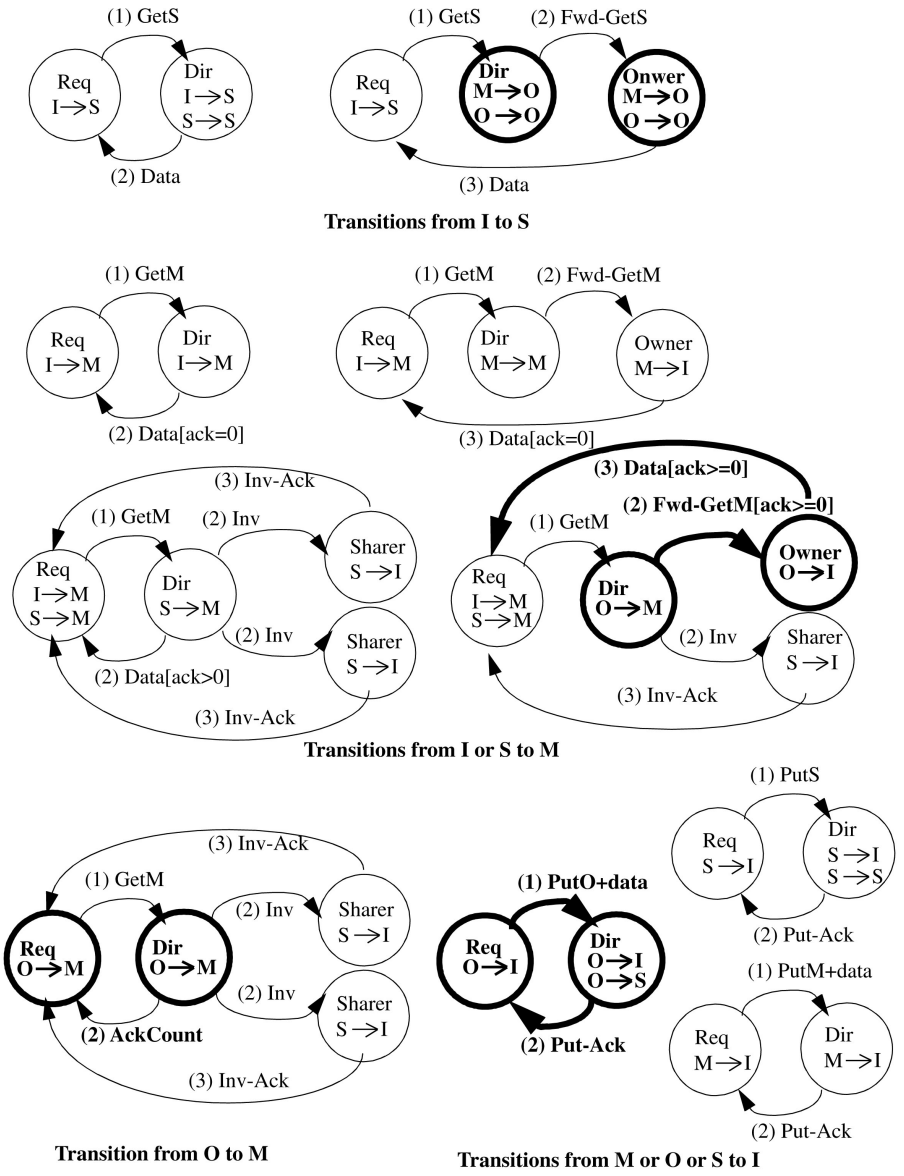

FIGURE 8.7: High-Level Description of MOSI Directory Protocol. In each transition, the cache controller that requests the transaction is denoted "Req".

and appends the AckCount. The directory also sends Invalidations to each of the sharers. The owner receives the Fwd-GetM and responds to the requestor with Data and the AckCount. The requestor uses this received AckCount to determine when it has received the last Inv-Ack. There is a similar transaction if the requestor of the GetM was the owner (in state O). The difference here is that the directory sends the AckCount directly to the requestor because the requestor is the owner.

This protocol has a PutO transaction that is nearly identical to the PutM transaction. It contains data for the same reason that the PutM transaction contains data, i.e., because both M and O are dirty states.

8.4.2 Detailed Protocol Specification

Tables 8.5 and 8.6 present the detailed specification of the MOSI protocol, including transient states. Differences with respect to the MSI protocol are highlighted with boldface font. The protocol adds to the set of cache states both the stable O state as well as transient OM^{AC}, OM^{A}, and

TABLE 8.5: MOSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	AckCount from Dir	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^{D}	send GetM to Dir/IM^{AD}											
IS^{D}	stall	stall	stall			stall		-/S		-/S			
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^{A}	-/M		ack--	
IM^{A}	stall	stall	stall	stall	stall							ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^{A}			send Inv-Ack to Req/I							
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^{A}	-/M		ack--	
SM^{A}	hit	stall	stall	stall	stall							ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^{A}	**send data to Req/O**	send data to Req/I								
MI^{A}	stall	stall	stall	**send data to Req/OI^{A}**	send data to Req/II^{A}		-/I						
O	**hit**	**send GetM to Dir/OM^{AC}**	**send PutO+data to Dir/OI^{A}**	**send data to Req**	**send data to Req/I**								
OM^{AC}	**hit**	**stall**	**stall**	**send data to Req**	**send data to Req/IM^{AD}**						**-/OM^{A}**	**ack--**	
OM^{A}	**hit**	**stall**	**stall**	**send data to Req**	**stall**							**ack--**	**-/M**
OI^{A}	**stall**	**stall**	**stall**	**send data to Req**	**send data to Req/II^{A}**		**-/I**						
SI^{A}	stall	stall	stall			send Inv-Ack to Req/II^{A}	-/I						
II^{A}	stall	stall	stall				-/I						

TABLE 8.6: MOSI Directory Protocol—Directory Controller

	GetS	GetM from Owner	GetM from NonOwner	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	PutO+data from Owner	PutO+data from NonOwner
I	send Data to Req, add Req to Sharers/S		send Data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req		send Put-Ack to Req
S	send Data to Req, add Req to Sharers		send Data to Req, send Inv to Sharers, set Owner to Req, clear Sharers/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I		remove Req from Sharers, send Put-Ack to Req		remove Req from Sharers, send Put-Ack to Req
O	forward GetS to Owner, add Req to Sharers	send AckCount to Req, send Inv to Sharers, clear Sharers/M	forward GetM to Owner, send Inv to Sharers, set Owner to Req, clear Sharers, send AckCount to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, copy data to mem, send Put-Ack to Req, clear Owner/S	remove Req from Sharers, send Put-Ack to Req	copy data to memory, send Put-Ack to Req, clear Owner/S	remove Req from Sharers, send Put-Ack to Req
M	forward GetS to Owner, add Req to Sharers/O		forward GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to mem, send Put-Ack to Req, clear Owne/I	send Put-Ack to Req		send Put-Ack to Req

OI^A states to handle transactions for blocks initially in state O. The state OM^{AC} indicates that the cache is waiting for both Inv-Acks (A) from caches and an AckCount (C) from the directory, but not data. Because this block started in state O, it already had valid data.

An interesting situation arises when core C1's cache controller has a block in OM^{AC} or SM^{AD} and receives a Fwd-GetM or Invalidation from core C2 for that block. C2's GetM must have been ordered at the directory before C1's GetM, for this situation to arise. Thus, the directory state changes to M (owned by C2) before observing C1's GetM. When C2's Fwd-GetM or Invalidation arrives at C1, C1 must be aware that C2's GetM was ordered first. Thus, C1's cache state changes from either OM^{AC} or SM^{AD} to IM^{AD}. The forwarded GetM or Invalidation from C2 invalidated C1's cache block and now C1 must wait for both Data and Inv-Acks.

8.5 REPRESENTING DIRECTORY STATE

In the previous sections, we have assumed a *complete directory*; that is, the directory maintains the complete state for each block, including the full set of caches that (may) have shared copies. Yet this assumption contradicts the primary motivation for directory protocols: scalability. In a system with a large number of caches (i.e., a large number of potential sharers of a block), maintaining the complete set of sharers for each block requires a significant amount of storage, even when using a compact bit-vector representation. For a system with a modest number of caches, it may be reason-

able to maintain this complete set, but the architects of larger-scale systems may wish for more scalable solutions to maintaining directory state.

There are many ways to reduce how much state the directory maintains for each block. Here we discuss two important techniques: coarse directories and limited pointers. We discuss these techniques independently, but observe that they can be combined. We contrast each solution with the baseline design, illustrated in the top entry of Figure 8.8.

8.5.1 Coarse Directory

Having the complete set of sharers enables the directory to send Invalidation messages to exactly those cache controllers that have the block in state S. One way to reduce the directory state is to conservatively maintain a coarse list of sharers that is a superset of the actual set of sharers. That is, a given entry in the sharer list corresponds to a set of K caches, as illustrated in the middle entry of Figure 8.8. If one or more of the caches in that set (may) have the block in state S, then that bit in the sharer list is set. A GetM will cause the directory controller to send an Invalidation to all K caches in that set. Thus, coarse directories reduce the directory state at the expense of extra interconnection network bandwidth for unnecessary Invalidation messages, plus the cache controller bandwidth to process these extra Invalidation messages.

8.5.2 Limited Pointer Directory

In a chip with C caches, a complete sharer list requires C entries, one bit each, for a total of C bits. However, studies have shown that many blocks have zero sharers or one sharer. A limited pointer directory exploits this observation by having i ($i<C$) entries, where each entry requires $log_2 C$ bits, for a total of $i * log_2 C$ bits, as illustrated in the bottom entry of Figure 8.8. A limited pointer directory

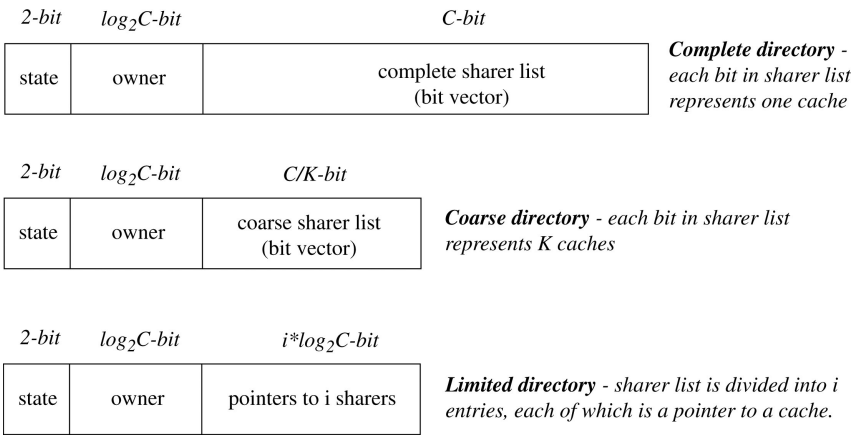

FIGURE 8.8: Representing directory state for a block in a system with N nodes.

requires some additional mechanism to handle (hopefully uncommon) situations in which the system attempts to add an $i+1^{th}$ sharer. There are three well-studied options for handling these situations, denoted using the notation Dir_iX [2, 8], where i refers to the number of pointers to sharers, and X refers to the mechanism for handling situations in which the system attempts to add an $i+1^{th}$ sharer.

- Broadcast (Dir_iB): If there are already i sharers and another GetS arrives, the directory controller sets the block's state to indicate that a subsequent GetM requires the directory to broadcast the Invalidation to all caches (i.e., a new "too many sharers" state). A drawback of Dir_iB is that the directory could have to broadcast to all C caches even when there are only K sharers ($i<K<C$), requiring the directory controller to send (and the cache controllers to process) $C-K$ unnecessary Invalidation messages. The limiting case, Dir_0B, takes this approach to the extreme by eliminating all pointers and requiring a broadcast on all coherence operations. The original Dir_0B proposal maintained two state bits per block, encoding the three MSI states plus a special "Single Sharer" state [3]. This new state helps eliminate a broadcast when a cache tries to upgrade its S copy to an M copy (similar to the Exclusive state optimization). Similarly, the directory's I state eliminates broadcast when memory owns the block. AMD's Coherent HyperTransport [6] implements a version of Dir_0B that uses no directory state, forgoing these optimizations but eliminating the need to store any directory state. All requests sent to the directory are then broadcast to all caches.
- No Broadcast (Dir_iNB): If there are already i sharers and another GetS arrives, the directory asks one of the current sharers to invalidate itself to make room in the sharer list for the new requestor. This solution can incur significant performance penalties for widely-shared blocks (i.e., blocks shared by more than i nodes), due to the time spent invalidating sharers. Dir_iNB is especially problematic for systems with coherent instruction caches because code is frequently widely shared.
- Software (Dir_iSW): If there are already i sharers and another GetS arrives, the system traps to a software handler. Trapping to software enables great flexibility, such as maintaining a full sharer list in software-managed data structures. However, because trapping to software incurs significant performance costs and implementation complexities, this approach has seen limited commercial acceptance.

8.6 DIRECTORY ORGANIZATION

Logically, the directory contains a single entry for every block of memory. Many traditional directory-based systems, in which the directory controller was integrated with the memory controller, directly implemented this logical abstraction by augmenting memory to hold the directory. For

example, the SGI Origin added additional DRAM chips to store the complete directory state with each block of memory [10].

However, with today's multicore processors and large LLCs, the traditional directory design makes little sense. First, architects do not want the latency and power overhead of a directory access to off-chip memory, especially for data cached on chip. Second, system designers balk at the large off-chip directory state when almost all memory blocks are not cached at any given time. These drawbacks motivate architects to optimize the common case by caching only a subset of directory entries on chip. In the rest of this section, we discuss directory cache designs, several of which were previously categorized by Marty and Hill [13].

Like conventional instruction and data caches, a directory cache [7] provides faster access to a subset of the complete directory state. Because directories summarize the states of coherent caches, they exhibit locality similar to instruction and data accesses, but need only store each block's coherence state rather than its data. Thus, relatively small directory caches achieve high hit rates. Directory caching has no impact on the functionality of the coherence protocol; it simply reduces the average directory access latency. Directory caching has become even more important in the era of multicore processors. In older systems in which cores resided on separate chips and/or boards, message latencies were sufficiently long that they tended to amortize the directory access latency. Within a multicore processor, messages can travel from one core to another in a handful of cycles, and the latency of an off-chip directory access tends to dwarf communication latencies and become a bottleneck. Thus, for multicore processors, there is a strong incentive to implement an on-chip directory cache to avoid costly off-chip accesses.

The on-chip directory cache contains a subset of the complete set of directory entries. Thus, the key design issue is handling directory cache misses, i.e., when a coherence request arrives for a block whose directory entry is not in the directory cache.

We summarize the design options in Table 8.7 and describe them next.

8.6.1 Directory Cache Backed by DRAM

The most straightforward design is to keep the complete directory in DRAM, as in traditional multi-chip multiprocessors, and use a separate directory cache structure to reduce the average access latency. A coherence request that misses in this directory cache leads to an access of this DRAM directory. This design, while straightforward, suffers from several important drawbacks. First, it requires a significant amount of DRAM to hold the directory, including state for the vast majority of blocks that are not currently cached on the chip. Second, because the directory cache is decoupled from the LLC, it is possible to hit in the LLC but miss in the directory cache, thus incurring a DRAM access even though the data is available locally. Finally, directory cache replacements must write the directory entries back to DRAM, incurring high latency and power overheads.

TABLE 8.7: Comparing Directory Cache Designs

		Inclusive directory caches (Section 8.6.2)				
	Directory cache backed by DRAM directory (Section 8.6.1)	**Inclusive directory cache embedded in inclusive LLC (Section 8.6.2.1)**		**Standalone inclusive directory cache (Section 8.6.2.2)**		**Null Directory Cache (Section 8.6.3)**
		No Recalls	**With Recalls**	**No Recalls**	**With Recalls**	
directory location	DRAM	LLC		LLC		none
uses DRAM	yes	no		no		no
miss at directory implies	must access DRAM	block must be I		block must be I		block could be in any state → must broadcast
inclusion requirements	none	LLC includes L1s		directory cache includes L1s		none
implementation costs	DRAM plus separate on-chip cache	larger LLC blocks; highly associative LLC	larger LLC blocks	highly associative storage for redundant tags	storage for redundant tags	none
replacement notification	none	none	desirable	required	desirable	none

8.6.2 Inclusive Directory Caches

We can design directory caches that are more cost-effective by exploiting the observation that we need only cache directory states for blocks that are being cached on the chip. We refer to a directory cache as an *inclusive directory cache* if it holds directory entries for a superset of all blocks cached on the chip. An inclusive directory cache serves as a "perfect" directory cache that never misses for accesses to blocks cached on chip. There is no need to store a complete directory in DRAM. A miss in an inclusive directory cache indicates that the block is in state I; a miss is *not* the precursor to accessing some backing directory store.

We now discuss two inclusive directory cache designs, plus an optimization that applies to both designs.

8.6.2.1 Inclusive Directory Cache Embedded in Inclusive LLC

The simplest directory cache design relies on an LLC that maintains *inclusion* with the upper-level caches. Cache inclusion means that if a block is in an upper-level cache then it must also be present in a lower-level cache. For the system model of Figure 8.1, LLC inclusion means that if a block is in a core's L1 cache, then it must also be in the LLC.

A consequence of LLC inclusion is that if a block is *not* in the LLC, it is also *not* in an L1 cache and thus must be in state I for all caches on the chip. An inclusive directory cache exploits this property by embedding the coherence state of each block in the LLC. If a coherence request is sent to the LLC/directory controller and the requested address is not present in the LLC, then the directory controller knows that the requested block is not cached on-chip and thus is in state I in all the L1s.

Because the directory mirrors the contents of the LLC, the entire directory cache may be embedded in the LLC simply by adding extra bits to each block in the LLC. These added bits can lead to non-trivial overhead, depending on the number of cores and the format in which directory state is represented. We illustrate the addition of this directory state to an LLC cache block in Figure 8.9, comparing it to an LLC block in a system without the LLC-embedded directory cache.

Unfortunately, LLC inclusion has several important drawbacks. First, while LLC inclusion can be maintained automatically for private cache hierarchies (if the lower-level cache has sufficient associativity [4]), for the shared caches in our system model, it is generally necessary to send special "Recall" requests to invalidate blocks from the L1 caches when replacing a block in the LLC (discussed further in Section 8.6.2.3). More importantly, LLC inclusion requires maintaining redundant copies of cache blocks that are in upper-level caches. In multicore processors, the collective capacity of the upper-level caches may be a significant fraction of (or sometimes, even larger than) the capacity of the LLC.

8.6.2.2 Standalone Inclusive Directory Cache

We now present an inclusive directory cache design that does not rely on LLC inclusion. In this design, the directory cache is a standalone structure that is logically associated with the directory controller, instead of being embedded in the LLC itself. For the directory cache to be inclusive, it must contain directory entries for the union of the blocks in all the L1 caches because a block in the LLC but not in any L1 cache must be in state I. Thus, in this design, the directory cache consists of duplicate copies of the tags at all L1 caches. Compared to the previous design (Section 8.6.2.1),

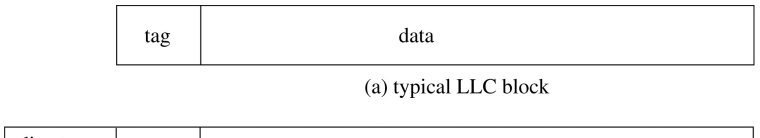

FIGURE 8.9: The cost of implementing the LLC-embedded directory cache

this design is more flexible, by virtue of not requiring LLC inclusion, but it has the added storage cost for the duplicate tags.

This inclusive directory cache has some significant implementation costs. Most notably, it requires a highly associative directory cache. (If we embed the directory cache in an inclusive LLC (Section 8.6.2.1), then the LLC must also be highly associative.) Consider the case of a chip with C cores, each of which has a K-way set-associative L1 cache. The directory cache must be C*K-way associative to hold all L1 cache tags, and the associativity unfortunately grows linearly with core count. We illustrate this issue for K=2 in Figure 8.10.

The inclusive directory cache design also introduces some complexity, in order to keep the directory cache up-to-date. When a block is evicted from an L1 cache, the cache controller must notify the directory cache regarding which block was replaced by issuing an explicit PutS request (e.g., we cannot use a protocol with a silent eviction, as discussed in Section 8.7.4). One common optimization is to piggy-back the explicit PutS on the GetS or GetX request. Since the index bits must be the same, the PutS can be encoded by specifying which way was replaced. This is sometimes referred to as a "replacement hint," although in general it is required (and not truly a "hint").

8.6.2.3 Limiting the Associativity of Inclusive Directory Caches

To overcome the cost of the highly-associative directory cache in the previous implementation, we present a technique for limiting its associativity. Rather than design the directory cache for the worst-case situation (C*K associativity), we limit the associativity by not permitting the worst-case to occur. That is, we design the directory cache to be A-way set associative, where A<C*K, and we do not permit more than A entries that map to a given directory cache set to be cached on chip. When a cache controller issues a coherence request to add a block to its cache, and the corresponding set in the directory cache is already full of valid entries, then the directory controller first evicts one of the blocks in this set from all caches. The directory controller performs this eviction by issu-

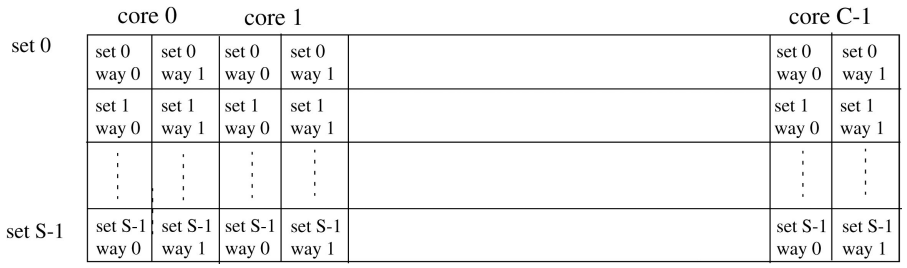

FIGURE 8.10: Inclusive directory cache structure (assumes 2-way L1 caches). Each entry is the tag corresponding to that set and way for the core at the top of the column.

ing a "Recall" request to all of the caches that hold this block in a valid state, and the caches respond with acknowledgments. Once an entry in the directory cache has been freed up via this Recall, then the directory controller can process the original coherence request that triggered the Recall.

The use of Recalls overcomes the need for high associativity in the directory cache but, without careful design, it could lead to poor performance. If the directory cache is too small, then Recalls will be frequent and performance will suffer. Conway et al. [6] propose a rule of thumb that the directory cache should cover at least the size of the aggregate caches it includes, but it can also be larger to reduce the rates of recalls. Also, to avoid unnecessary Recalls, this scheme works best with non-silent evictions of blocks in state S. With silent evictions, unnecessary Recalls will be sent to caches that no longer hold the block being recalled.

8.6.3 Null Directory Cache (with no backing store)

The least costly directory cache is to have no directory cache at all. Recall that the directory state helps prune the set of coherence controllers to which to forward a coherence request. But as with Coarse Directories (Section 8.5.1), if this pruning is done incompletely, the protocol still works correctly, but unnecessary messages are sent and the protocol is less efficient than it could be. Taken to the extreme, a Dir_0B protocol (Section 8.5.2) does no pruning whatsoever, in which case it does not actually need a directory at all (or a directory cache). Whenever a coherence request arrives at the directory controller, the directory controller simply forwards it to all caches (i.e., broadcasts the forwarded request). This directory cache design, which we call the Null Directory Cache, may seem simplistic, but it is popular for small- to medium-scale systems because it incurs no storage cost.

One might question the purpose of a directory controller if there is no directory state, but it serves two important roles. First, as with all other systems in this chapter, the directory controller is responsible for the LLC; it is, more precisely, an LLC/directory controller. Second, the directory controller serves as an ordering point in the protocol; if multiple cores concurrently request the same block, the requests are ordered at the directory controller. The directory controller resolves which request happens first.

8.7 PERFORMANCE AND SCALABILITY OPTIMIZATIONS

In this section, we discuss several optimizations to improve the performance and scalability of directory protocols.

8.7.1 Distributed Directories

So far we have assumed that there is a single directory attached to a single monolithic LLC. This design clearly has the potential to create a performance bottleneck at this shared, central resource.

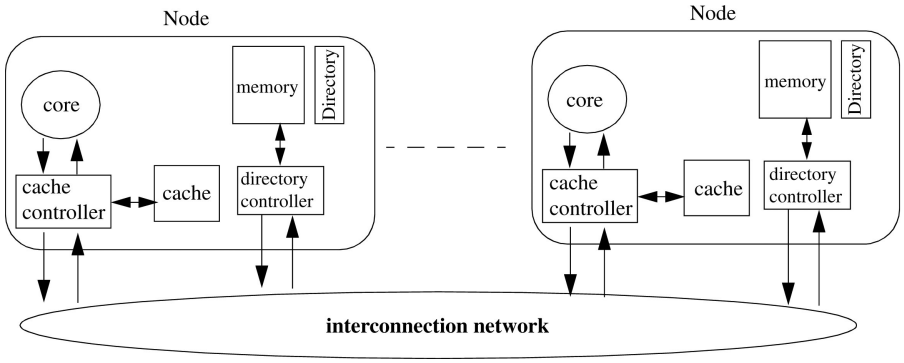

FIGURE 8.11: Multiprocessor system model with distributed directory.

The typical, general solution to the problem of a centralized bottleneck is to distribute the resource. The directory for a given block is still fixed in one place, but different blocks can have different directories.

In older, multi-chip multiprocessors with N *nodes*—each node consisting of multiple chips, including the processor core and memory—each node typically had 1/N of the memory associated with it and the corresponding $1/N^{th}$ of the directory state.

We illustrate such a system model in Figure 8.11. The allocation of memory addresses to nodes is often static and often easily computable using simple arithmetic. For example, in a system with N directories, block B's directory entry might be at directory *B modulo N*. Each block has a *home*, which is the directory that holds its memory and directory state. Thus, we end up with a system in which there are multiple, independent directories managing the coherence for different sets of blocks. Having multiple directories provides greater bandwidth of coherence transactions than requiring all coherence traffic to pass through a single, central resource. Importantly, distributing the directory has no impact on the coherence protocol.

In today's world of multicore processors with large LLCs and directory caches, the approach of distributing the directory is logically the same as in the traditional multi-chip multiprocessors. We can distribute (bank) the LLC and directory cache. Each block has a home bank of the LLC with its associated bank of the directory cache.

8.7.2 Non-Stalling Directory Protocols

One performance limitation of the protocols presented thus far is that the coherence controllers stall in several situations. In particular, the cache controllers stall when they receive forwarded requests for blocks in certain transient states, such as IM^A. In Tables 8.8 and 8.9, we present a variant of the baseline MSI protocol that does not stall in these scenarios. For example, when a cache

TABLE 8.8: Non-stalling MSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			send Inv-Ack to Req/IS^DI		-/S		-/S		
IS^DI	**stall**	**stall**	**stall**					**-/I**		**-/I**		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	**-/IM^AS**	**-/IM^AI**						ack--	-/M
IM^AS	**stall**	**stall**	**stall**			**send Inv-Ack to Req/IM^ASI**					**ack--**	**send data to Req and Dir/S**
IM^ASI	**stall**	**stall**	**stall**								**ack--**	**send data to Req and Dir/I**
IM^AI	**stall**	**stall**	**stall**								**ack--**	**send data to Req/I**
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	**-/SM^AS**	**-/SM^AI**						ack--	-/M
SM^AS	**stall**	**stall**	**stall**			**send Inv-Ack to Req/SM^ASI**					**ack--**	**send data to Req and Dir/S**
SM^ASI	**stall**	**stall**	**stall**								**ack--**	**send data to Req and Dir/I**
SM^AI	**stall**	**stall**	**stall**								**ack--**	**send data to Req/I**
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

controller has a block in state IM^A and receives a Fwd-GetS, it processes the request and changes the block's state to IM^AS. This state indicates that after the cache controller's GetM transaction completes (i.e., when the last Inv-Ack arrives), the cache controller will change the block state to S. At this point, the cache controller must also send the block to the requestor of the GetS and to the directory, which is now the owner. By not stalling on the Fwd-GetS, the cache controller can improve performance by continuing to process other forwarded requests behind that Fwd-GetS in its incoming queue.

TABLE 8.9: Non-stalling MSI Directory Protocol—Directory Controller

	GetS	GetM	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data
I	send data to Req, add Req to Sharers/S	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req	
S	send data to Req, add Req to Sharers	send data to Req, set Owner to Req, send Inv to Sharers, clear Sharers/M	send Put-Ack to Req, remove Req from Sharers	send Put-Ack to Req, remove Req from Sharers/I		remove Req from Sharers, send Put-Ack to Req	
M	forward GetS to Owner, add Req to Sharers, clear Owner/S^D	forward GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to memory, send Put-Ack to Req, clear Owner/I	send Put-Ack to Req	
S^D	stall	stall	send Put-Ack to Req, remove Req from Sharers	send Put-Ack to Req, remove Req from Sharers		remove Req from Sharers, send Put-Ack to Req	copy data to memory/S

A complication in the non-stalling protocol is that, while in state IM^AS, a Fwd-GetM could arrive. Instead of stalling, the cache controller processes this request and changes the block's state to IM^ASI (in I, going to M, waiting for Inv-Acks, then will go to S and then to I). A similar set of transient states arises for blocks in SM^A. Removing stalling leads to more transient states, in general, because the coherence controller must track (using new transient states) the additional messages it is processing instead of stalling.

We did not remove the stalls from the directory controller. As with the memory controllers in the snooping protocols in Chapter 7, we would need to add an impractically large number of transient states to avoid stalling in all possible scenarios.

8.7.3 Interconnection Networks Without Point-to-Point Ordering

We mentioned in Section 8.2, when discussing the system model of our baseline MSI directory protocol, that we assumed that the interconnection network provides point-to-point ordering for the Forwarded Request network. At the time, we claimed that point-to-point ordering simplifies the architect's job in designing the protocol because ordering eliminates the possibility of certain races.

We now present one example race that is possible if we do not have point-to-point ordering in the interconnection network. We assume the MOSI protocol from Section 8.4. Core C1's cache owns a block in state M. Core C2 sends a GetS request to the directory and core C3 sends a GetM request to the directory. The directory receives C2's GetS first and then C3's GetM. For both requests, the directory forwards them to C1.

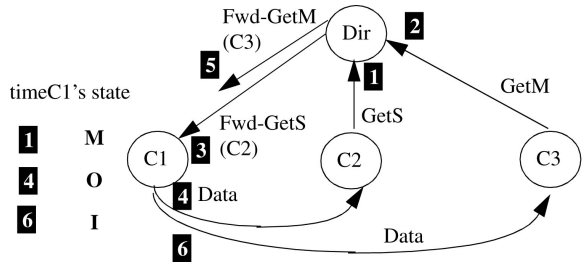

FIGURE 8.12: Example with point-to-point ordering.

- With point-to-point ordering (illustrated in Figure 8.12): C1 receives the Fwd-GetS, responds with Data, and changes the block state to O. C1 then receives the Fwd-GetM, responds with Data, and changes the block state to I. This is the expected outcome.
- Without point-to-point ordering (illustrated in Figure 8.13): The Fwd-GetM from C3 may arrive at C1 first. C1 responds with Data to C3 and changes the block state to I. The Fwd-GetS from C2 then arrives at C1. C1 is in I and cannot respond. The GetS request from C2 will never be satisfied and the system will eventually deadlock.

The directory protocols we have presented thus far are not compatible with interconnection networks that do not provide point-to-point order for the Forwarded Request network. To make the protocols compatible, we would have to modify them to correctly handle races like the one described above. One typical approach to eliminating races like these is to add extra handshaking messages. In the example above, the directory could wait for the cache controller to acknowledge reception of each forwarded request sent to it before forwarding another request to it.

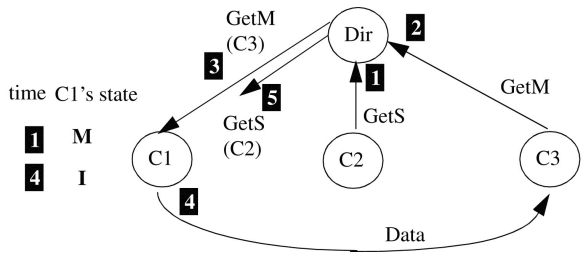

FIGURE 8.13: Example without point-to-point ordering. Note that C2's Fwd-GetS arrives at C1 in state I and thus C1 does not respond.

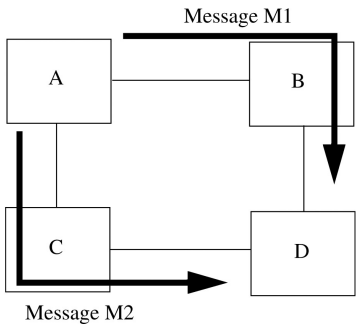

FIGURE 8.14: Adaptive Routing Example.

Given that point-to-point ordering reduces complexity, it would seem an obvious design decision. However, enforcing point-to-point ordering prohibits us from implementing some potentially useful optimizations in the interconnection network. Notably, it prohibits the unrestricted use of adaptive routing.

Adaptive routing enables a message to dynamically choose its path as it traverses the network, generally to avoid congested links or switches. Adaptive routing, although useful for spreading traffic and mitigating congestion, enables messages between endpoints to take different paths and thus arrive in a different order than that in which they were sent. Consider the example in Figure 8.14, in which Switch A sends two messages, M1 and then M2, to Switch D. With adaptive routing, they take different paths, as shown in the figure. If there happens to be more congestion at Switch B than at Switch C, then M2 could arrive at Switch D before M1, despite being sent after M1.

8.7.4 Silent vs. Non-Silent Evictions of Blocks in State S

We designed our baseline directory protocol such that a cache cannot silently evict a block in state S (i.e., without issuing a PutS to notify the directory). To evict an S block, the cache must send a PutS to the directory and wait for a Put-Ack. Another option would be to allow silent evictions of S blocks. (A similar discussion could be made for blocks in state E, if one considers the E state to not be an ownership state, in which case silent evictions of E blocks are possible.)

Advantages of Silent PutS

The drawback to the explicit PutS is that it uses interconnection network bandwidth—albeit for small data-free PutS and Put-Ack messages—even in cases when it ends up not being helpful. For example if core C1 sends a PutS to the directory and then subsequently wants to perform a load to this block, C1 sends a GetS to the directory and re-acquires the block in S. If C1 sends this second

GetS before any intervening GetM requests from other cores, then the PutS transaction served no purpose but did consume bandwidth.

Advantages of Explicit PutS

The primary motivation for sending a PutS is that a PutS enables the directory to remove the cache no longer sharing the block from its list of sharers. There are three benefits to having a more precise sharer list. First, when a subsequent GetM arrives, the directory need not send an Invalidation to this cache. The GetM transaction is accelerated by eliminating the Invalidation and having to wait for the subsequent Inv-Ack. Second, in a MESI protocol, if the directory is precisely counting the sharers, it can identify situations in which the last sharer has evicted its block; when the directory knows there are no sharers, it can respond to a GetS with Exclusive data. Third, recall from Section 8.6.2 that directory caches that use Recalls can benefit from having explicit PutS messages to avoid unnecessary Recall requests.

A secondary motivation for sending a PutS, and the reason our baseline protocol does send a PutS, is that it simplifies the protocol by eliminating some races. Notably, without a PutS, a cache that silently evicts a block in S and then sends a GetS request to re-obtain that evicted block in S can receive an Invalidation from the directory before receiving the data for its GetS. In this situation, the cache does not know if the Invalidation pertains to the first period in which it held the block in S or the second period (i.e., whether the Invalidation is serialized before or after the GetS). The simplest solution to this race is to pessimistically assume the worst case (the Invalidation pertains to the second period) and always invalidate the block as soon as its data arrives. More efficient solutions exist, but complicate the protocol.

8.8 CASE STUDIES

In this section, we discuss several commercial directory coherence protocols. We start with a traditional multi-chip system, the SGI Origin 2000. We then discuss more recently developed directory protocols, including AMD's Coherent HyperTransport and the subsequent HyperTransport Assist. Last, we present Intel's QuickPath Interconnect (QPI).

8.8.1 SGI Origin 2000

The Silicon Graphics Origin 2000 [10] was a commercial multi-chip multiprocessor designed in the mid-1990s to scale to 1024 cores. The emphasis on scalability necessitated a scalable coherence protocol, resulting in one of the first commercial shared-memory systems using a directory protocol. The Origin's directory protocol evolved from the design of the Stanford DASH multiprocessor [11], as the DASH and Origin had overlapping architecture teams.

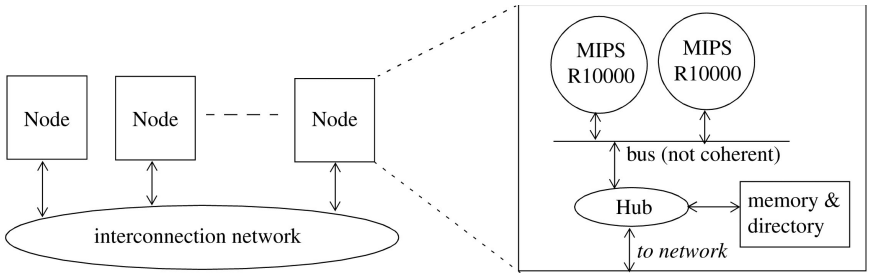

FIGURE 8.15: SGI Origin.

As illustrated in Figure 8.15, the Origin consists of up to 512 nodes, where each node consists of two MIPS R10000 processors connected via a bus to a specialized ASIC called the Hub. Unlike similar designs, Origin's processor bus does not exploit coherent snooping and simply connects the processors to each other and to the node's Hub. The Hub manages the cache coherence protocol and interfaces the node to the interconnection network. The Hub also connects to the node's portion of the distributed memory and directory. The network does not support any ordering, even point-to-point ordering between nodes. Thus, if Processor A sends two messages to Processor B, they may arrive in a different order than that in which they were sent.

The Origin's directory protocol has a few distinguishing features that are worth discussing. First, because of its scalability, each directory entry contains fewer bits than necessary to represent every possible cache that could be sharing a block. The directory dynamically chooses, for each directory entry, to use either a coarse bit vector representation or a limited pointer representation (Section 8.5).

A second interesting feature in the protocol is that because the network provides no ordering, there are several new coherence message race conditions that are possible. Notably, the examples from Section 8.7.3 are possible. To maintain correctness, the protocol must consider all of these possible race conditions introduced by not enforcing ordering in the network.

A third interesting feature is the protocol's use of a non-ownership E state. Because the E state is not an ownership state, a cache can silently evict a block in state E (or state S). The Origin provides a special Upgrade coherence request to transition from S to E without needlessly requesting data, which is not unusual but does introduce a new race. There is a window of vulnerability between when processor P1 sends an Upgrade and when the Upgrade is serialized at the directory; if another processor's GetM or Upgrade is serialized first, then P1's state is I when its Upgrade arrives at the directory, and P1 in fact needs data. In this situation, the directory sends a negative acknowledgment (NACK) to P1, and P1 must send a GetM to the directory.

Another interesting feature of the Origin's E state is how requests are satisfied when a pro-

cessor is in E. Consider the case where processor P1 obtains a block in state E. If P2 now sends a GetS to the directory, the directory must consider that P1 (a) might have silently evicted the block, (b) might have an unmodified value of the block (i.e., with the same value as at memory), or (c) might have a modified value of the block. To handle all of these possibilities, the directory responds with data to P2 and also forwards the request to P1. P1 sends P2 either new data (if in M) or just an acknowledgment. P2 must wait for both responses to arrive to know which message's data to use.

One other quirk of the Origin is that it uses only two networks (request and response) instead of the three required to avoid deadlock. A directory protocol has three message types (request, forwarded request, and response) and thus nominally requires three networks. Instead, the Origin protocol detects when deadlock could occur and sends a "backoff" message to a requestor on the response network. The backoff message contains the list of nodes that the request needs to be sent to, and the requestor can then send to them on the request network.

8.8.2 Coherent HyperTransport

Directory protocols were originally developed to meet the needs of highly scalable systems, and the SGI Origin is a classic example of such a system. Recently, however, directory protocols have become attractive even for small- to medium-scale systems because they facilitate the use of point-to-point links in the interconnection network. This advantage of directory protocols motivated the design of AMD's Coherent HyperTransport (HT) [5]. Coherent HT enables glueless connections of AMD processors into small-scale multiprocessors. Perhaps ironically, Coherent HT actually uses broadcasts, thus demonstrating that the appeal of directory protocols in this case is the use of point-to-point links, rather than scalability.

AMD observed that systems with up to eight processor chips can be built with only three point-to-point links per chip and a maximum chip-to-chip distance of three links. Eight processor chips, each of which can have 6-cores in current generation technology, means a system with a respectable 48 cores. To keep the protocol simple, Coherent HT uses a variation on a Dir_0B directory protocol (Section 8.5.2) that stores no stable directory state. Any coherence request sent to the directory is forwarded to all cache controllers (i.e., broadcast). Coherent HT can also be thought of as an example of a null directory cache: requests always miss in the (null) directory cache, so it always broadcasts. Because of the broadcasts, the protocol does not scale to large-scale systems, but that was not the goal.

In a system with Coherent HT, each processor chip contains some number of cores, one or more integrated memory controllers, one or more integrated HyperTransport controllers, and between one and three Coherent HT links to other processor chips. A "node" consists of a processor chip and its associated memory for which it is the home.

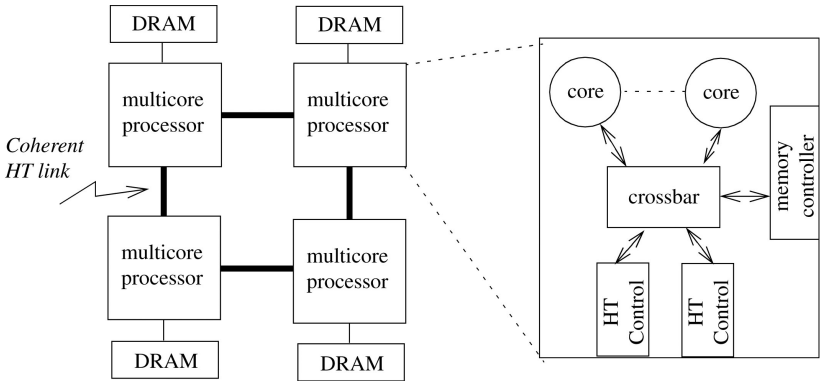

FIGURE 8.16: Four-node Coherent HyperTransport System (adapted from [5]).

There are many viable interconnection network topologies, such as the four-node system shown in Figure 8.16. Significantly, this protocol does not require a total order of coherence requests, which provides greater flexibility for the interconnection network.

A coherence transaction works as follows. A core unicasts a coherence request to the directory controller at the home node, as in a typical directory protocol. Because the directory has no state and thus cannot determine which cores need to observe the request, the directory controller then broadcasts the forwarded request to all cores, including the requestor. (This broadcast is like what happens in a snooping protocol, except that the broadcast is not totally ordered and does not originate with the requestor.) Each core then receives the forwarded request and sends a response (either data or an acknowledgment) to the requestor. Once the requestor has received all of the responses, it sends a message to the directory controller at the home node to complete the transaction.

Looking at this protocol, one can view it as the best or worst of both worlds. Optimistically, it has point-to-point links with no directory state or complexity, and it is sufficiently scalable for up to eight processors. Pessimistically, it has the long three-hop latency of directories—or four hops, if you consider the message from the requestor to the home to complete the transaction, although this message is not on the critical path—with the high broadcast traffic of snooping. In fact, Coherent HT uses even more bandwidth than snooping because all broadcasted forwarded requests generate a response. The drawbacks of Coherent HT motivated an enhanced design called HyperTransport Assist [6].

8.8.3 HyperTransport Assist

For the 12-core AMD Opteron processor-based system code-named Magny Cours, AMD developed HyperTransport Assist [6]. HT Assist enhances Coherent HT by eliminating the need to

broadcast every forwarded coherence request. Instead of the Dir_0B-like protocol of Coherent HT, HT Assist uses a directory cache similar to the design described in Section 8.6.2. Each multicore processor chip has an inclusive directory cache that has a directory entry for every block (a) for which it is the home *and* (b) that is cached anywhere in the system. There is no DRAM directory, thus preserving one of the key features of Coherent HT. A miss in the directory cache indicates that the block is not cached anywhere. HT Assist's directory cache uses Recall requests to handle situations in which the directory cache is full and needs to add a new entry. Although HT Assist appears from our description thus far to be quite similar to the design in Section 8.6.2, it has several distinguishing features that we present in greater detail.

First, the directory entries provide only enough information to determine whether a coherence request must be forwarded to all cores, forwarded to a single core, or satisfied by the home node's memory. That is, the directory does not maintain sufficient state to distinguish needing to forward a request to two cores from needing to forward the request to all cores. This design decision eliminated the storage cost of having to maintain the exact number of sharers of each block; instead, two directory states distinguish "one sharer" from "more than one sharer."

Second, the HT Assist design is careful to avoid incurring a large number of Recalls. AMD adhered to a rule of thumb that there should be at least twice as many directory entries as cached blocks. Interestingly, AMD chose not to send explicit PutS requests; their experiments apparently convinced them that the additional PutS traffic was not worth the limited benefit in terms of a reduction in Recalls.

Third, the directory cache shares the LLC. The LLC is statically partitioned by the BIOS at boot time, and the default is to allocate 1MB to the directory cache and allocate the remaining 5MB to the LLC itself. Each 64-byte block of the LLC that is allocated to the directory cache is interpreted as 16 4-byte directory entries, organized as four 4-way set-associative sets.

8.8.4 Intel QPI

Intel developed its QuickPath Interconnect (QPI) [9, 12] for connecting processor chips starting with the 2008 Intel Core microarchitecture, and QPI first shipped in the Intel Core i7-9xx processor. Prior to this, Intel connected processor chips with a shared-wire bus called the Front-Side Bus (FSB). FSB evolved from a single shared bus to multiple buses, but the FSB approach was fundamentally bottlenecked by the electrical signaling limits of the buses. To overcome this limitation, Intel designed QPI to connect processor chips with point-to-point (i.e., non-bus) links. QPI specifies multiple levels of the networking stack, from physical layer to protocol layer. For purposes of this primer, we focus on the protocol layer here.

QPI supports five stable coherence states, the typical MESI states and the F(orward) state. The F state is a clean, read-only state, and it is distinguished from the S state because a cache with

a block in F may respond with data (i.e., forward the data) to coherence requests. Only one cache may hold a block in F at a given time. The F state is somewhat similar to the O state, but differs in that a block in F is not dirty and can thus be silently evicted; a cache that wishes to evict a block in O must copy the block back to memory. The benefit of the F state is that it allows read-only data to be sourced from a cache, which is often faster than sourcing it from memory (which usually responds to requests when a block is read-only).

QPI provides two different protocol modes, depending on the size of the system: "home snoop" and "source snoop."

QPI's Home Snoop mode is effectively a scalable directory protocol (i.e., do not be confused by the word "snoop" in its name[2]). As with typical directory protocols, a core C1 issues a request to the directory at the home node C2, and the directory forwards that request to only the node(s) that need to see it, say C3 (the owner in M). C3 responds with data to C1 and also sends a message to C2 to notify the directory. When the directory at C2 receives the notification from C3, it sends a "completion" message to C1, at which point C1 may use the data it received from C3. The directory serves as the serialization point in the protocol and resolves message races.

QPI's Source Snoop protocol mode is designed to have lower-latency coherence transactions, at the expense of not scaling well to large systems with many nodes. A core C1 broadcasts a request to all nodes, including the home. Each core responds to the home with a "snoop response" that indicates what state the block was in at that core; if the block was in state M, then the core sends the block to the requestor in addition to the snoop response to the home. Once the home has received all of the snoop responses for a request, the request has been ordered. At this point, the home either sends data to the requestor (if no core owned the block in M) or a non-data message to the requestor; either message, when received by the requestor, completes the transaction.

Source Snoop's use of broadcast requests is similar to a snooping protocol, but with the critical difference of the broadcast requests not traveling on a totally ordered broadcast network. Because the network is not totally ordered, the protocol must have a mechanism to resolve races (i.e., when two broadcasts race, such that core C1 sees broadcast A before broadcast B and core C2 sees B before A). This mechanism is provided by the home node, albeit in a way that differs from typical race ordering in directory protocols. Typically, the directory at the requested block's home orders two racing requests based on which request arrives at the home first. QPI's Source Snoop, instead, orders the requests based on which request's snoop responses have all arrived at the home.

Consider the race situation in which block A is initially in state I in the caches of both C1 and C2. C1 and C2 both decide to broadcast GetM requests for A (i.e., send a GetM to the other core and to the home). When each core receives the other core's GetM, it sends a snoop response

[2] Intel uses the word "snoop" to refer to what a core does when it receives a coherence request from another node.

to the home. Assume that C2's snoop response arrives at the home before C1's snoop response. In this case, C1's request is ordered first and the home sends data to C1 and informs C1 that there is a race. C1 then sends an acknowledgment to the home, and the home subsequently sends a message to C1 that both completes C1's transaction and tells C1 to send the block to C2. Handling this race is somewhat more complicated than in a typical directory protocol in which requests are ordered when they arrive at the directory.

Source Snoop mode uses more bandwidth than Home Snoop, due to broadcasting, but Source Snoop's common case (no race) transaction latency is less. Source Snoop is somewhat similar to Coherent HyperTransport, but with one key difference. In Coherent HT, a request is unicasted to the home, and the home broadcasts the request. In Source Snoop, the requestor broadcasts the request. Source Snoop thus introduces more complexity in resolving races because there is no single point at which requests can be ordered; Coherent HT uses the home for this purpose.

8.9 DISCUSSION AND THE FUTURE OF DIRECTORY PROTOCOLS

Directory protocols have come to dominate the market. Even in small-scale systems, directory protocols are more common that snooping protocols, largely because they facilitate the use of point-to-point links in the interconnection network. Furthermore, directory protocols are the *only* option for systems requiring scalable cache coherence. Although there are numerous optimizations and implementation tricks that can mitigate the bottlenecks of snooping, fundamentally none of them can eliminate these bottlenecks. For systems that need to scale to hundreds or even thousands of nodes, a directory protocol is the only viable option for coherence. Because of their scalability, we anticipate that directory protocols will continue their dominance for the foreseeable future.

It is possible, though, that future highly scalable systems will not be coherent or at least not coherent across the entire system. Perhaps such systems will be partitioned into subsystems that are coherent, but coherence is not maintained across the subsystems. Or perhaps such systems will follow the lead of supercomputers, like those from Cray, that have either not provided coherence [14] or have provided coherence but restricted what data can be cached [1].

8.10 REFERENCES

[1] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003. doi:10.1109/IPDPS.2003.1213087

[2] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 280–89, May 1988. doi:10.1109/ISCA.1988.5238

[3] J. K. Archibald and J.-L. Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 355–62, June 1984. doi:10.1145/800015.808205

[4] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 73–80, May 1988. doi:10.1109/ISCA.1988.5212

[5] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, March/April 2007. doi:10.1109/MM.2007.43

[6] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March/April 2010. doi:10.1109/MM.2010.31

[7] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proceedings of the International Conference on Parallel Processing*, 1990.

[8] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–18, Nov. 1993. doi:10.1145/161541.161544

[9] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. Document Number 320412-001US, Jan. 2009.

[10] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 241–51, June 1997.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992. doi:10.1109/2.121510

[12] R. A. Maddox, G. Singh, and R. J. Safranek. *Weaving High Performance Multiprocessor Fabric: Architecture Insights into the Intel QuickPath Interconnect*. Intel Press, 2009.

[13] M. R. Marty and M. D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[14] S. L. Scott. Synchronization and Communication in the Cray T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–36, Oct. 1996.

• • • •