8

Directory-Based Cache Coherence

This chapter examines an important part of the development of parallel architectures: putting together cache coherence and a scalable, distributed-memory machine organization. We have studied cache coherence for bus-based machines with centralized memory. We have also seen that in order to scale up machines, memory is distributed, a scalable point-to-point interconnection network is introduced, and a communication assist provides varying degrees of interpretation of network transactions to support programming models. Regardless of the sophistication of that assist, all of the scalable machines we have studied have the generic structure depicted in Figure 8.1.

At the final point in our design spectrum so far, the communication assist provides a shared address space in hardware. However, while the natural inclination of caches is to replicate referenced data in a shared address space, we have not yet examined how cache coherence may be provided. In fact, to avoid the coherence problem and simplify memory consistency, the machines in that final design point disable the hardware caching of logically shared but physically remote data, restricting the programming model.

This chapter takes on the important issue of how implicit caching and coherence may be provided in hardware on a machine with physically distributed memory, without the benefits of a globally snoopable interconnect such as a bus. Not only must the hardware latency and bandwidth scale well, as we have seen, but so must the protocols used for coherence, at least up to the scales of practical interest. We focus on full hardware support for cache coherence and particularly on the most common approach called directory-based cache coherence. In terms of the layers of abstraction, the shared address space programming model with coherent replication is supported directly at the hardware/software interface, as shown in Figure 8.2. Other programming models, such as message passing, can be implemented in software. The next chapter describes some alternative approaches that take different positions on hardware/software trade-offs, such as coherent replication in main memory rather than in the caches, coherence under software control, and alternative memory consistency models.

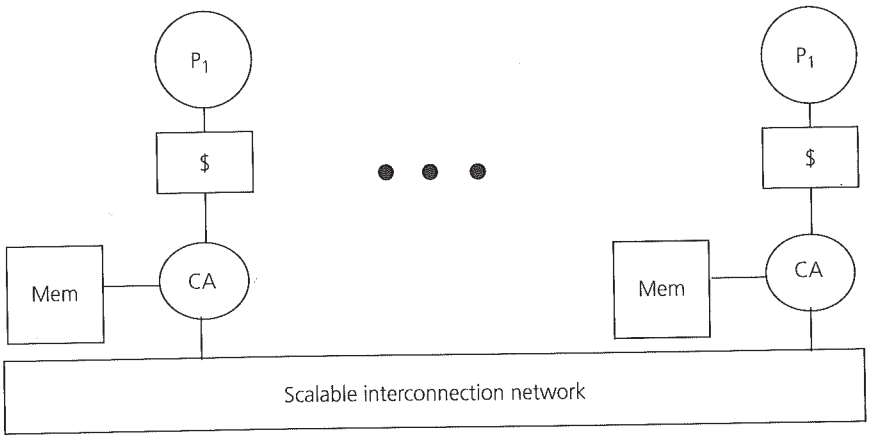


FIGURE 8.1 A generic scalable multiprocessor. This diagram represents the generic structure of the machines discussed in Chapter 7: processing nodes with physically distributed memory and a scalable interconnect. The processing nodes may be uniprocessors (as shown) or multiprocessors.

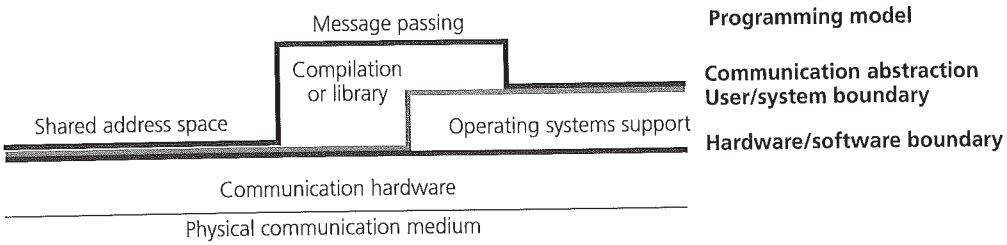


FIGURE 8.2 Layers of abstraction for systems discussed in this chapter. A coherent, shared physical address space is supported directly in hardware and message passing through software layers.

Scalable cache coherence is typically based on the concept of a directory. Since the state of a block in the caches can no longer be determined implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain this state explicitly in a place—called a *directory*—where requests can go and look it up. Consider a simple example. Imagine that each cache-line-sized block of main memory has associated with it a record of the caches that currently contain a copy of the block and the state of the block in those caches. This record is called the *directory entry* for that block (see Figure 8.3). As in bus-based systems, there may be many caches with a clean, readable block, but if the block is writable (possibly modified) in one cache, then only that cache may have a valid copy. When a node incurs a cache miss, it first communicates with the directory entry for the block using point-to-point network transactions. Since the directory entry is colocated with the

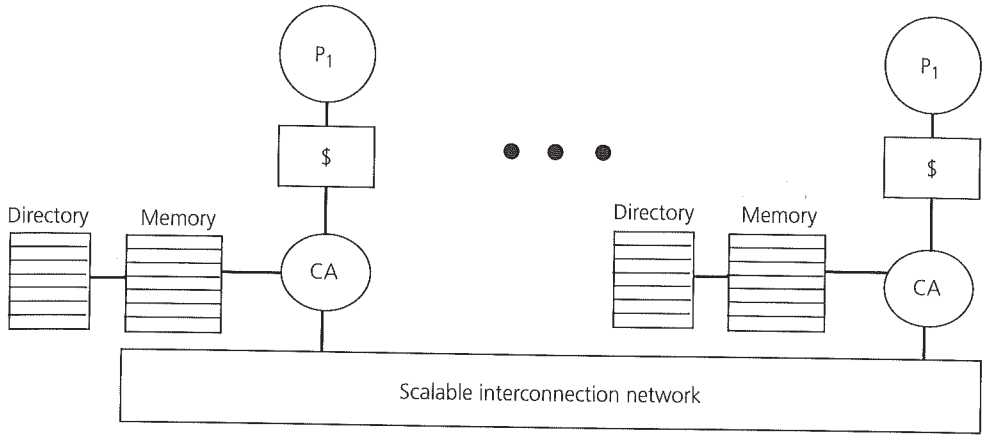


FIGURE 8.3 A scalable multiprocessor with directories. Every block of main memory, the size of a cache block, has a directory entry that keeps track of its cached copies and their state.

main memory for the block, its location can be determined from the address of the block. From the directory, the node determines where the valid cached copies (if any) are and what further actions to take. It then communicates with the cached copies as necessary using additional network transactions. For example, it may obtain a modified block from another node or, on a write operation, send invalidations to other nodes and receive acknowledgments from them. The resulting changes to the states of cached blocks are also communicated to the directory entry through network transactions, so the directory stays up-to-date.

In a directory protocol, requests, replies, invalidations, updates, and acknowledgments across nodes are all network transactions like those of the previous chapter, only here the endpoint processing at the destination of the transaction (invalidating blocks, retrieving and replying with data) is typically done by the communication assist rather than the main processor. (As in previous chapters, we will call response transactions that carry data "replies" and all others simply "responses.") Since directory schemes rely on point-to-point network transactions, they can be used with any interconnection network. Important questions for directories include the form in which the directory information is stored and how correct, efficient protocols may be designed using these representations.

While directories constitute the dominant approach to scalable cache coherence, other approaches can be contemplated. One approach that has been tried is to extend the broadcast and snooping mechanism, using a hierarchy of broadcast media like buses or rings. This is conceptually attractive because it builds larger systems hierarchically out of existing small-scale mechanisms. However, it does not apply to general network topologies such as meshes and cubes, and we will see that it has problems with latency and bandwidth, so it has not become very popular. An

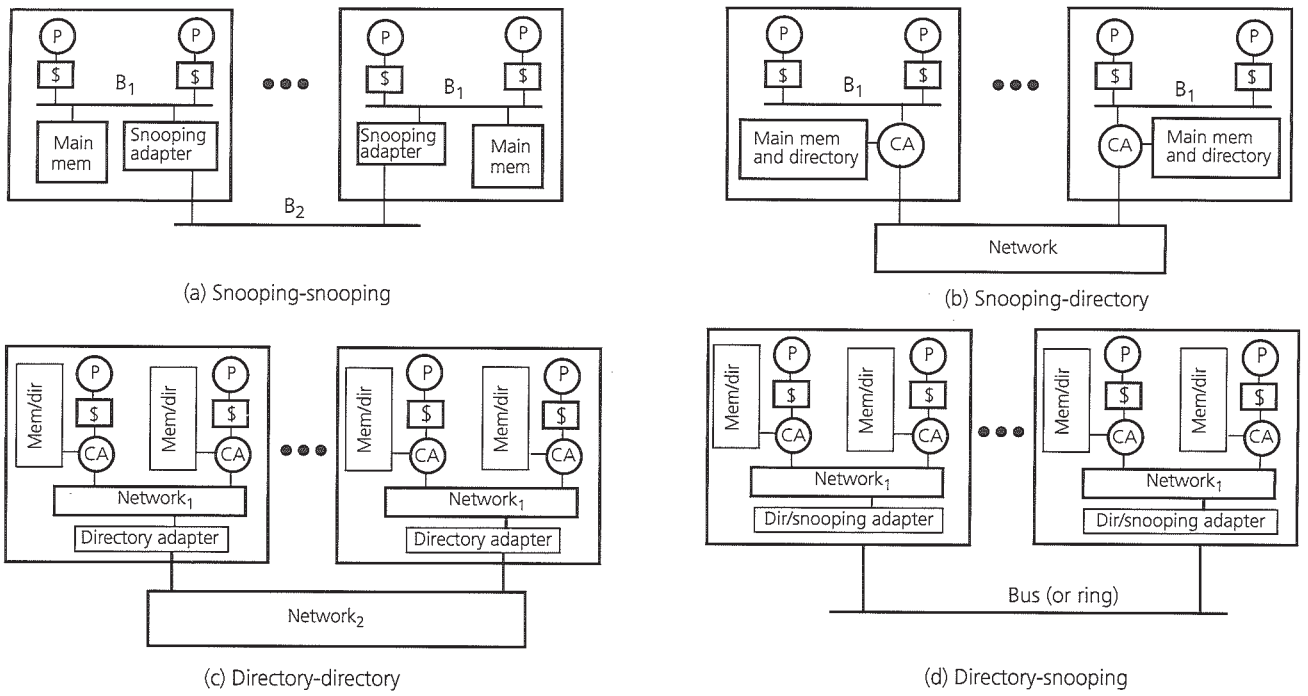
approach that is popular is a limited, two-level protocol hierarchy. Each node of the machine is itself a multiprocessor. The caches within a node are kept coherent by one coherence protocol called the *inner protocol*. Coherence across nodes is maintained by another, possibly different protocol called the *outer protocol*. To the outer protocol, each multiprocessor node looks like a single cache, and coherence within the node is the responsibility of the inner protocol. Usually, an adapter or a shared tertiary cache is used to represent a node to the outer protocol. A common organization is for the outer protocol to be a directory protocol and the inner one to be a snooping protocol (Lovett and Clapp 1996; Lenoski et al. 1993; Clark and Alnes 1996; Weber et al. 1997). However, other combinations such as snooping-snooping (Frank, Burkhardt, and Rothnie 1993), directory-directory (Convex Computer Corporation 1993), and even snooping-directory may be used (see Figure 8.4).

Putting together smaller-scale machines to build larger machines in a two-level organization is an attractive engineering option: it amortizes fixed per-node costs over the processors in a node, may take advantage of packaging hierarchies, and may satisfy much of the interprocessor communication less expensively within a node. The main focus of this chapter will be on directory protocols across nodes, regardless of whether the node is a uni- or multiprocessor or what coherence method it uses. The interactions among two-level protocols are also discussed. While we focus on directory protocols because they have been most successful and are likely to remain the most popular, we will briefly examine the less popular hierarchical approaches as well. As we examine the organizational structure of the directory, the protocols used to support coherence and consistency, and the requirements placed on the communication assist, we will find another rich and interesting design space.

The first section of this chapter presents a framework for understanding the different approaches to providing coherent replication in a shared address space, including snooping, directories, and hierarchical snooping. Section 8.2 introduces the basic operation of a directory protocol using a simple directory representation and then provides an overview of alternative directory organizations and protocols. This is followed by a quantitative assessment of some high-level issues and architectural trade-offs for directory protocols in Section 8.3.

The next few sections cover the issues and techniques involved in actually designing correct, efficient protocols. Section 8.4 discusses the major new challenges introduced by the presence of multiple copies of data without a serializing interconnect. The next two sections delve deeply into the two most popular types of directory-based protocols, discussing various design alternatives and using two commercial architectures as case studies: the Origin2000 from Silicon Graphics, Inc. and the NUMA-Q from Sequent Computer Systems, Inc. Section 8.7 examines the impact of key performance parameters of the communication architecture on the end performance of parallel programs under directory protocols.

Synchronization for directory-based multiprocessors is discussed in Section 8.8 and the implications for parallel software in Section 8.9. Section 8.10 covers some advanced topics, including the approaches of hierarchically extending snooping and directory protocols for scalable coherence.



(a) Snooping-snooping

(b) Snooping-directory

(c) Directory-directory

(d) Directory-snooping

FIGURE 8.4 Some possible organizations for two-level cache-coherent systems. Each node visible at the outer level is itself a multiprocessor. B_1 is a first-level bus, and B_2 is a second-level bus. CA is the communication assist. The label snooping-directory, for example, means that a snooping protocol is used to maintain coherence within a multiprocessor node, and a directory protocol is used to maintain coherence across nodes.

8.1 SCALABLE CACHE COHERENCE

This section briefly lays out the major organizational alternatives for providing coherent replication in a multiprocessor's extended memory hierarchy and introduces the basic mechanisms that any approach to coherence must provide.

On a machine with physically distributed memory, nonlocal data may be replicated either only in the processors' caches or in the local main memory. If coherent replication is provided in main memory, additional support for keeping caches coherent may not be necessary since only data that is already coherent in the local main memory may enter the cache. This chapter assumes that data is automatically replicated only in the caches, not in main memory, and that it is kept coherent in hardware at the granularity of cache blocks, just as in bus-based machines. Since main memory is physically distributed and has nonuniform access costs to a processor, architectures of this type are often called *cache-coherent, nonuniform memory access* or CC-NUMA architectures. More generally, systems that provide a shared address space programming model with physically distributed memory and coherent replication (either in caches or main memory) are called *distributed shared memory* (DSM) systems.

Any approach to coherence, including the snooping coherence discussed in Chapters 5 and 6, must provide certain critical mechanisms. First, a block can be in each cache (or local replication store) in one of a number of states, potentially in different states in different caches. The protocol must provide these cache states as well as the state transition diagram, according to which blocks in different caches independently change states, and the set of actions associated with the state transition diagram. Directory-based protocols also have a *directory state* for each block, which is the state of the block as known to the directory. The protocol may be invalidation based, update based, or hybrid, and the stable cache states themselves are very often the same (e.g., MESI), regardless of whether the system is based on snooping or directories. The trade-offs in the choices of stable cache states are very similar to those discussed in Chapter 5 and are not revisited in this chapter. Conceptually, for any protocol, the cache state of a memory block is a vector containing its state in every cache in the system. The same state transition diagram governs the copies in different caches, though the current state of the block at any given time may be different in different caches. The state changes for a block in different caches are coordinated through transactions on the interconnect, whether bus transactions or more general network transactions.

Given a protocol at the cache state transition level, a coherent system must provide mechanisms for managing the protocol. First, a mechanism is needed to determine when (i.e., on which operations) to invoke the protocol. This is done in the same way on most systems: through an *access fault* (cache miss) detection mechanism. The protocol is invoked if the processor makes an access that its cache cannot satisfy by itself, for example, an access to a block that is not in the cache or a write access to a block that is present but in shared state. However, even when they use the same set of cache states, transitions, and access fault mechanisms, approaches to

cache coherence differ substantially in the mechanisms they provide for three important functions that may need to be performed when an access fault occurs:

1. Finding out enough information about the state of the location (cache block) in other caches to determine what action to take
2. Locating those other copies, if needed (e.g., to invalidate them)
3. Communicating with the other copies (e.g., obtaining data from them or invalidating or updating them)

In snooping protocols, all three functions are performed by the broadcast and snooping mechanism. The processor puts a "search" request on the bus, containing the address of the block, and other cache controllers snoop and respond. It is possible to use a broadcast and "snooping" method in distributed machines as well; the assist at the node incurring the miss can broadcast messages to all nodes, and their assists can examine the incoming request and respond as appropriate. However, broadcast does not scale since it generates a large amount of traffic (at least p network transactions on every miss on a p-node machine). Scalable approaches include hierarchical snooping and directory-based approaches.

In a hierarchical snooping approach, the interconnection network is not a single broadcast bus (or ring) but a tree of buses. The processors are in the bus-based snooping multiprocessors at the leaves of the tree. Parent buses are connected to children by interfaces that snoop the buses on both sides and propagate relevant transactions upward or downward in the hierarchy. Main memory may be centralized at the root or distributed among the leaves. In this case, all of the preceding functions are performed by the hierarchical extension of the broadcast and snooping mechanism: a processor puts a search request on its bus as before, and it is propagated up and down the hierarchy as necessary based on snoop results. The hope is that most of the time a request will not have to be propagated very far. Hierarchical snooping systems are discussed further in Section 8.10.2.

In the simple directory approach introduced earlier in the chapter, information about the state of blocks in other caches is found by looking up the directory through network transactions. The location of the copies is also found from the directory, and the copies are communicated with using point-to-point network transactions in an arbitrary interconnection network, without resorting to broadcast. How the directory information is actually organized influences how protocols might be structured around this organization using network transactions and, hence, how the protocol addresses the three key functions required for coherence.

8.2 OVERVIEW OF DIRECTORY-BASED APPROACHES

This section begins by more fully describing a simple directory scheme and how it might operate using cache states, directory states, and network transactions. It then discusses the organizational issues in scaling directories to large numbers of nodes, provides a classification of scalable directory organizations, and discusses the basics of protocols associated with these organizations.

The following definitions are useful for our discussion of directory protocols. For a given cache or memory block:

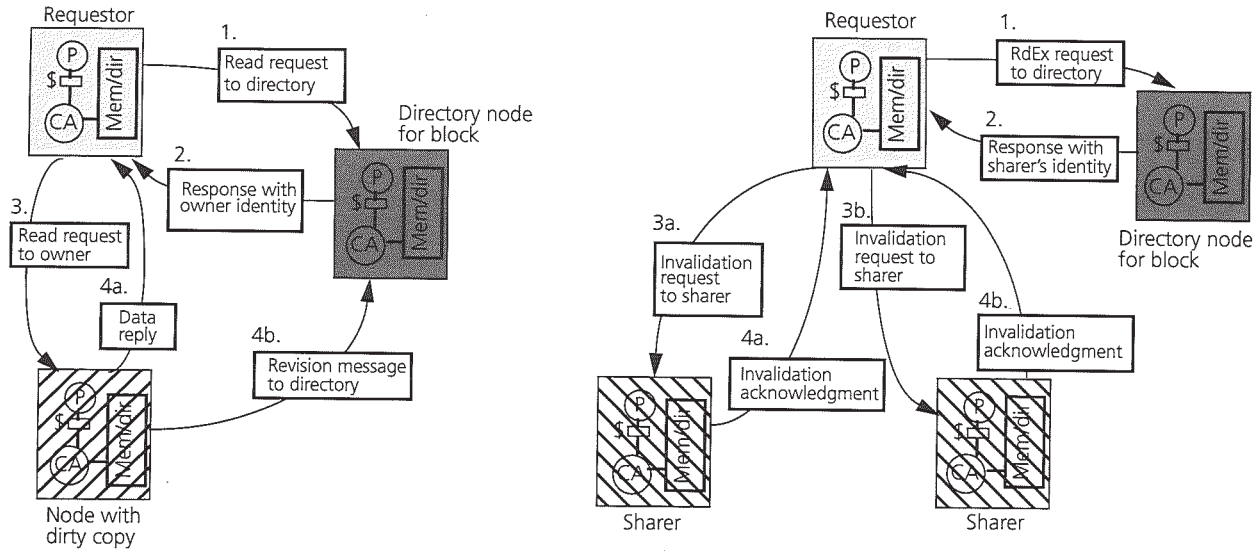
- The *home node* is the node in whose main memory the block is allocated.
- The *dirty node* is the node that has a copy of the block in its cache in modified (dirty) state. Note that the home node and the dirty node for a block may be the same.
- The *owner node* is the node that currently holds the valid copy of a block and must supply the data when needed; in directory protocols, this is either the home node (when the block is not in dirty state in a cache) or the dirty node.
- The *exclusive node* is the node that has a copy of the block in its cache in an exclusive state, either dirty or (clean) exclusive as the case may be. (Recall from Chapter 5 that the cache state called exclusive means this is the only valid cached copy and that the block in main memory is up-to-date.) Thus, the dirty node is also the exclusive node.
- The *local node*, or *requesting node*, is the node containing the processor that issues a request for the block.
- Blocks whose home is local to the issuing processor are called *locally allocated* or simply *local blocks*, whereas all others are called *remotely allocated* or *remote blocks*.

Let us begin with the basic operation of directory-based protocols, using a very simple directory organization.

8.2.1 Operation of a Simple Directory Scheme

When a cache miss (access control fault) is incurred, the local node sends a request network transaction to the home node where the directory information for the block is located. On a read miss, the directory indicates from which node the data may be obtained, as shown in Figure 8.5(a). On a write miss, the directory identifies the copies of the block, and invalidation or update network transactions may be sent to these copies (Figure 8.5[b]). (Recall that a write to a block in shared state is also considered a write miss.) Since invalidations or updates are sent to multiple copies through potentially disjoint paths in the network, determining the completion or commitment of a write now requires that all copies reply to invalidations with explicit acknowledgment transactions; we cannot assume completion or commitment when the read-exclusive or update request obtains access to the interconnect as we did on a shared bus since we cannot guarantee ordering with respect to other transactions within the interconnect.

A natural way to organize a directory is to maintain the directory information for a block together with the block in main memory, that is, at the home node for the block. A simple organization for the directory information for a block is as a bit vector of p *presence bits*—which indicate for each of the p nodes (uniprocessor or multiprocessor) whether that node has a cached copy of the block—together with one or more state bits (see Figure 8.6). Let us assume for simplicity that there is only one state bit, called the *dirty* bit, which indicates if the block is dirty in one of the node



(a) Read miss to a block in modified state in a cache

(b) Write miss to a block with two sharers

FIGURE 8.5 Basic operation of a simple directory. Two example operations are shown. On the left is a read miss to a block that is currently held in modified (dirty) state by a node that is not the requestor or the node that holds the directory information. A read miss to a block that is clean in main memory (i.e., at the directory node) is simpler: the main memory simply replies to the requestor with the data, and the miss is satisfied in a single request-response pair of transactions. On the right is a write miss to a block that is currently in shared state in two other nodes' caches (the two sharers). The big rectangles are the nodes, and the arcs (with boxed labels) are network transactions. The numbers 1, 2, and so on next to a transaction show the serialization of transactions. Different letters next to the same number indicate that the transactions can be performed in parallel and hence overlapped.

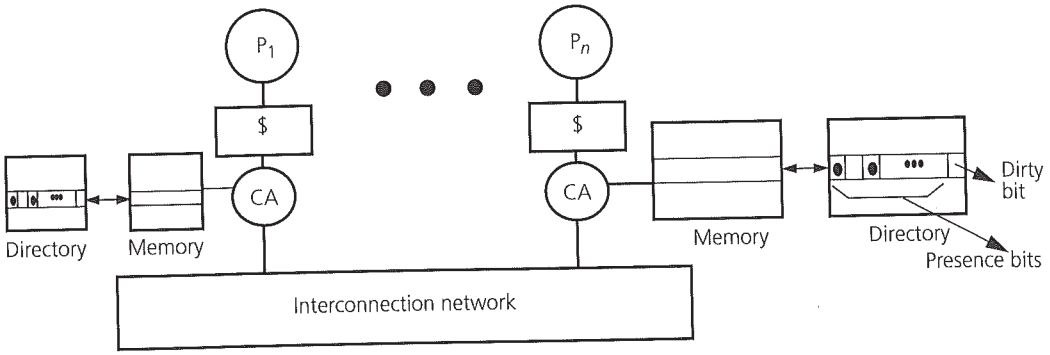


FIGURE 8.6 Directory information for a distributed-memory multiprocessor. In simple organization, the directory entry for a block is a vector of p presence bits, one for each node, and a dirty bit indicating whether any node has the block in modified state.

caches. Of course, if the dirty bit is ON, then only one node (the dirty node) should be caching that block and only that node's presence bit should be ON. With this structure, a read miss can easily determine from looking up the directory which node, if any, has a dirty copy of the block or if the block is valid in main memory at the home, and a write miss can determine which nodes are the sharers that must be invalidated.

The directory information for a block is simply main memory's view of the cache state of that block in different caches. The directory does not necessarily need to know the exact state (e.g., MESI) in each cache but only enough information to determine what actions to take. The number of states at the directory is therefore typically smaller than the number of cache states. In fact, since the directory and the caches communicate through a distributed interconnect, there will be periods when a directory's knowledge of a cache state is incorrect since the cache state has been modified but notice of the modification has not reached the directory. During this time, the directory may send a message to the cache based on its old (no longer valid) knowledge. The race conditions caused by this distribution of state make directory protocols interesting, and we see how they are handled using transient states or other means in Sections 8.4 through 8.6.

To see in greater detail how a read miss and write miss might interact with this bit vector directory organization, consider a protocol with three stable cache states (MSI), a single level of cache per processor, and a single processor per node. The protocol is orchestrated by the assists, which are also referred to as *coherence controllers* or *directory controllers*. On a read miss or a write miss at node i (including an upgrade from shared state), the local communication assist or controller looks up the address of the memory block to determine if the home is local or remote. If it is remote, a network transaction is sent to the home node for the block. There, the directory entry for the block is looked up, and the assist at the home may treat the miss as follows, using network transactions similar to those that were shown in Figure 8.5 (other, more optimized treatments are discussed later in the chapter):

- If the dirty bit is OFF, then the assist obtains the block from main memory, supplies it to the requestor in a reply network transaction, and turns the *i*th presence bit, *presence[i]*, ON.
- If the dirty bit is ON, then the assist responds to the requestor with the identity of the node whose presence bit is ON (i.e., the owner or dirty node). The requestor then sends a request network transaction to that owner node. At the owner, the cache changes its state to shared and supplies the block to both the requesting node, which stores the block in its cache in shared state, as well as to main memory at the home node. At memory, the dirty bit is turned OFF, and *presence[i]* is turned ON.

A write miss by processor *i* goes to memory and is handled as follows:

- If the dirty bit is OFF, then main memory has a clean copy of the data. Invalidation request transactions must be sent to all nodes *j* for which *presence[j]* is ON. Assuming a strict request-response scenario, as in Figure 8.5, the home node supplies the block to the requesting node *i* together with the presence bit vector. The directory entry is cleared, leaving only *presence[i]* and the dirty bit ON. (If the request is an upgrade instead of a read exclusive, an acknowledgment containing the bit vector is returned to the requestor instead of the data itself.) The assist at the requestor sends invalidation requests to the required nodes and waits for invalidation acknowledgment transactions from the nodes, indicating that the write has completed with respect to them. Finally, the requestor places the block in its cache in dirty state.
- If the dirty bit is ON, then the block is first recalled from the dirty node (whose presence bit is ON), using network transactions with the home and the dirty node. That cache changes its state to invalid, and then the block is supplied to the requesting processor, which places the block in its cache in dirty state. The directory entry is cleared, leaving only *presence[i]* and the dirty bit ON.

On a replacement of a dirty block by node *i*, the dirty data being replaced is written back to main memory, and the directory is updated to turn off the dirty bit and *presence[i]*. (As in bus-based machines, write backs cause interesting race conditions that are discussed later in the context of real protocols.) Finally, if a block in shared state is replaced from a cache, a message may or may not be sent to the directory to turn off the corresponding presence bit so an invalidation is not sent to this node the next time the block is written. This message is called a *replacement hint*; whether it is sent or not does not affect the correctness of the protocol or the execution.

A directory scheme similar to this one was introduced as early as 1978 (Censier and Feautrier 1978). It was designed for use in systems with a few processors and a centralized main memory and was used in the S-1 multiprocessor project at Lawrence Livermore National Laboratories (Widdoes and Correll 1980). However, directory schemes in one form or another were in use even before this. The earliest scheme was used in IBM mainframes, which had a few processors connected to a centralized memory through a high-bandwidth switch rather than a bus. With no broadcast medium to snoop on, a duplicate copy of the cache tags for each processor was maintained at the main memory, and it served as the directory. Requests coming

to the memory looked up all the tags to determine the states of the block in the different caches (Tang 1976; Tucker 1986). Of course, the tag copies at main memory had to be kept up-to-date. Since the directory was centralized in these early schemes, they are called *centralized directory schemes.*

The value of directories is that they keep track of which nodes have copies of a block, eliminating the need for broadcast. This is clearly very valuable on read misses since a request for a block will either be satisfied at the main memory or the directory will tell it exactly where to go to retrieve the exclusive copy. On write misses, the value of directories over the simpler broadcast approach is greatest if the number of sharers of the block (to which invalidations or updates must be sent) is usually small and does not scale up quickly with the number of processing nodes.

We might already expect the typical number of sharers to be small from our understanding of parallel applications. For example, in a near-neighbor grid computation, usually two, and at most four, processes should share a block at a partition boundary, regardless of the grid size or the number of processors. Even when a location is actively read and written by all processes in an application, the number of sharers to be invalidated at a write depends upon the temporal interleaving of reads and writes by processors. A common example is *migratory* data, which is read and written by one processor, then read and written by another processor, and so on (for example, a global sum into which processes accumulate their values). Although all processors read and write the location, only one other processor on a write—the previous writer—has a valid copy and must be invalidated since all others were invalidated before the previous write.

Empirical measurements of program behavior show that the number of valid copies on most writes to shared data is indeed very small the vast majority of the time, that this number does not grow quickly with the number of processors used, and that the frequency of writes that generate many invalidations is very low. Such data for our parallel applications will be presented and analyzed in light of application characteristics in Section 8.3.1. (Note that even if all processors running the application have to be invalidated on most writes, directories are still valuable for writes if the application does not run on all nodes of the multiprocessor.) These facts are also promising for the scalability of directory-based approaches and help us understand how to organize directories cost-effectively.

8.2.2 Scaling

The main goal of using directory protocols is to allow cache coherence to scale beyond the number of processors that may be sustained by a bus. It is important to understand the scalability of directory protocols in terms of both performance and the storage overhead for directory information. A system with distributed memory and interconnect already provides good scalability of raw latency and bandwidth under well-behaved loads. The major performance scaling issues for a protocol are how the latency and bandwidth demands it presents to the system scale with the number of processors used. The bandwidth demands are governed by the number of network transactions generated per miss (multiplied by the frequency of misses) and

latency by the number of these transactions that are in the critical path of the miss. In turn, these quantities are affected both by the directory organization and by how well the flow of network transactions is optimized in the protocol (given a directory organization). Storage, however, is affected only by how the directory information is organized. For the simple bit vector organization, the number of presence bits needed scales linearly with both the number of processing nodes (p bits per memory block) and the amount of main memory (1 bit vector per memory block), leading to a potentially large storage overhead for the directory. With a 64-byte block size and 64 processors, the directory storage overhead as a fraction of nondirectory (i.e., data) memory is 64 bits (plus state bits) divided by 64 bytes, or 12.5%, which is not so bad. With 256 processors and the same block size, the overhead is 50%, and with 1,024 processors it is 200%! The directory overhead does not scale well, though it may be acceptable if the number of nodes visible to the directory at the target machine scale is not very large.

Fortunately, there are many other ways to organize directory information that improve the scalability of directory storage. The different organizations naturally lead to different high-level protocols with different ways of addressing the three protocol functions presented in Section 8.1 and different performance characteristics. The rest of this section lays out the space of directory organizations and briefly describes how individual read and write misses might be handled in straightforward protocols that use these organizations. The discussion assumes that no other cache misses are in progress at the time, hence no race conditions, so the directory and the caches are always encountered as being in stable states. Deeper protocol issues are discussed in Sections 8.4 through 8.6.

8.2.3 Alternatives for Organizing Directories

Since communication with cached copies is always done through network transactions, the real differentiation among approaches is in the first two functions of coherence protocols: finding the source of the directory information upon a miss and determining the locations of the relevant copies.

The two major classes of alternatives for finding the source of the directory information for a block are known as *flat directory schemes* and *hierarchical directory schemes*.

The simple directory scheme described earlier is a flat scheme. Flat schemes are so named because the source of the directory information for a block is in a fixed place, usually at the home that is determined from the address of the block; on a miss, a single request network transaction is sent directly to the home node to look up the directory (if the home is remote) regardless of how far away the home is.

In hierarchical schemes, the source of directory information is not known a priori. Memory is again distributed with the processors, but the directory information for each block is logically organized as a hierarchical data structure (a tree). The processing nodes, each with its portion of memory, are at the leaves of the tree. The internal nodes of the tree are simply hierarchically maintained directory information for the block: a node keeps track of whether each of its children has a copy of a

block. Upon a miss, the directory information for the block is found by traversing up the hierarchy level by level through network transactions until a directory node is reached that indicates its subtree has the block in the appropriate state. Thus, a processor that misses simply sends a search message up to its parent, and so on, rather than directly to the home node for the block with a single network transaction. The directory tree for a block is logical, not necessarily physical, and can be embedded in any general interconnection network. Every block has its own logical directory tree. In fact, in practice, every processing node in the system not only serves as a leaf node for the blocks it contains but also stores directory information as an internal tree node for other blocks.

In the hierarchical case, the information about locations of copies is also maintained through the hierarchy itself; copies are found and communicated with by traversing up and down the hierarchy guided by directory information. For example, a directory entry at a node may indicate not only whether its subtree has valid copies of the block but also if copies of blocks allocated within its subtree may exist beyond its subtree. In flat schemes, how this information about copies is stored varies considerably. At the highest level, flat schemes can be divided into two classes: memory-based schemes and cache-based schemes. *Memory-based schemes* store the directory information about all cached copies at the home node of the block. The basic bit vector scheme described earlier is memory based: the locations of all copies are discovered at the home, and they can be communicated with directly through point-to-point messages. In *cache-based schemes,* the information about cached copies is not all contained at the home but is distributed among the copies themselves. The home simply contains a pointer to one cached copy of the block. Each cached copy then contains a pointer to (or the identity of) the node that has the next cached copy of the block, in a distributed linked-list organization. The locations of copies are therefore determined by traversing this list via network transactions.

Figure 8.7 summarizes the taxonomy. Hierarchical directories have some potential advantages. For example, a read miss to a block whose home is far away in the interconnection network topology might be satisfied closer to the issuing processor if another copy is found nearby as the request traverses up and down the hierarchy, instead of going all the way to the home. In addition, requests from different nodes can potentially be combined at a common ancestor in the hierarchy, with only one request sent on from there. These advantages depend on how well the logical hierarchy matches the underlying physical network topology. However, instead of only a few point-to-point network transactions needed to satisfy a miss in many flat schemes, the number of network transactions needed to traverse up and down the hierarchy can be much larger, which tends to have much greater impact on performance than distance traversed in the network (since the endpoint cost of initiating and handling network transactions dominates the per-hop cost). Each transaction along the way needs to look up (and perhaps modify) the directory information at its destination node, making transactions more expensive. As a result, the latency and bandwidth characteristics of hierarchical directory schemes tend to be much worse than for flat schemes, and these organizations are not popular on modern systems. Hierarchical directories are not, therefore, discussed much in this chapter but

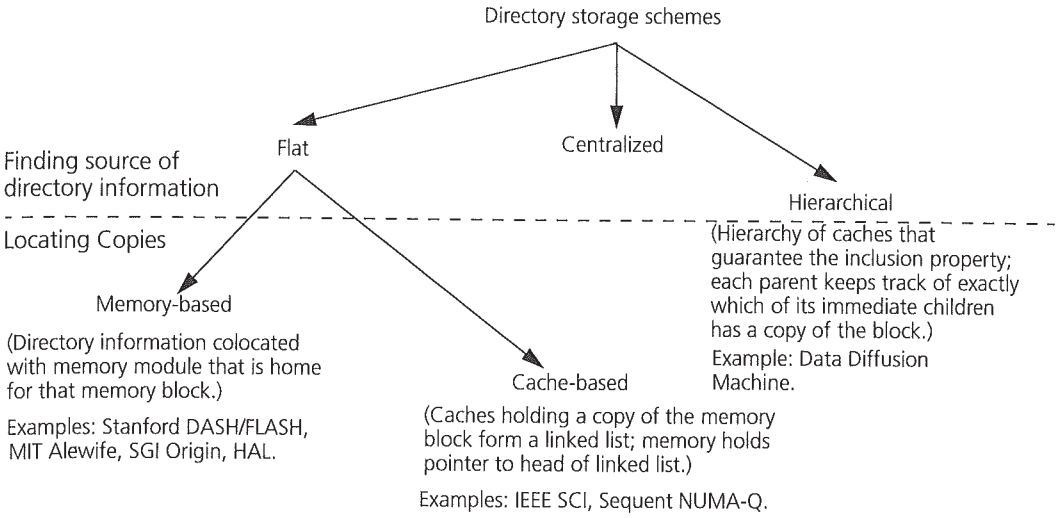


FIGURE 8.7 Alternatives for storing directory information. The two-level taxonomy is based on how the source of directory information and the copies themselves are located. In the hierarchical case, the same mechanism performs both functions.

are described briefly together with hierarchical snooping approaches in Section 8.10.2. The rest of this section examines flat directory schemes, both memory based and cache based, looking at directory organizations, storage overhead, the structure of protocols, and the impact on performance characteristics.

Flat, Memory-Based Directory Schemes

The bit vector organization described earlier, called a *full bit vector* organization, is the most straightforward way to store directory information in a flat, memory-based scheme. The style of protocol that results has already been discussed. Consider its basic performance characteristics on writes. Since it preserves information about sharers precisely and at the home, the number of network transactions per invalidating write grows only with the number of actual sharers. Because the identity of all sharers is available at the home, invalidations sent to them can be overlapped or even sent in parallel; the number of fully serialized network transactions in the critical path is thus not proportional to the number of sharers, reducing latency.

The main disadvantage of full bit vector schemes, as discussed earlier, is storage overhead. There are two ways to reduce this overhead for a given number of processors while still using full bit vectors. The first is to increase the cache block size. The second is to put multiple processors, rather than just one, in a node that is visible to the directory protocol; that is, to use a two-level protocol. For example, the Stanford DASH machine uses a full bit vector scheme, and its nodes are four-processor bus-based multiprocessors. These two methods actually make full bit vector directories

quite attractive for even fairly large machines: using four-processor nodes and 128-byte cache blocks, the directory memory overhead for a 256-processor machine is only 6.25%. As small-scale multiprocessors become increasingly attractive building blocks, this storage problem may not be severe.

However, these methods reduce the overhead by only a small constant factor each. The total directory storage is still proportional to $P*M$, where P is the number of processing nodes and M is the number of total memory blocks in the machine ($M = P*m$, where m is the number of blocks per local memory), and would become intolerable in very large machines. The overhead can be reduced further by addressing each of the factors in the $P*M$ expression. We can reduce the number of bits per directory entry, or *directory width,* by not letting it grow proportionally to P. Or we can reduce the total number of directory entries, or *directory height*, by not having an entry per memory block.

Directory width is reduced by using what are called *limited pointer directories,* which are motivated by the earlier observation that most of the time only a few caches have a copy of a block when the block is written. Limited pointer schemes therefore do not store yes or no information for all nodes but simply maintain a fixed number of pointers (say, i), each pointing to a node that currently caches a copy of the block (Agarwal et al. 1988). Each pointer takes $\log P$ bits of storage for P nodes, but the number of pointers used is small. For example, for a machine with 1,024 nodes, each pointer needs 10 bits, so even having 100 pointers uses less storage than a full bit vector scheme. In practice, five or less pointers seem to suffice. Of course, these schemes need some kind of backup or overflow strategy for the situation when more than i readable copies are cached since they can keep track of only i copies precisely. One strategy is to resort to broadcasting invalidations to all nodes when there are more than i copies. Many other strategies have been developed to avoid broadcast even in these cases. Different limited pointer schemes differ primarily in their overflow strategies and in the number of pointers they use.

Directory height can be reduced by organizing the directory itself as a cache, taking advantage of the fact that since the total amount of cache in the machine is much smaller than the total amount of memory, only a very small fraction of the memory blocks will actually be present in caches at a given time, so most of the directory entries will be unused anyway (Gupta, Weber, and Mowry 1990; O'Krafka and Newton 1990). Section 8.10 discusses techniques reducing directory width and height in more detail.

Regardless of these storage-reducing optimizations, the basic approach to finding copies and communicating with them (protocol functions [2] and [3]) remains the same for the different flat, memory-based schemes. The identities of the sharers are maintained at the home and (at least when there is no overflow) the copies are communicated with by sending point-to-point transactions to each.

Flat, Cache-Based Directory Schemes

In flat, cache-based schemes, there is still a home main memory for the block; however, the directory entry at the home node does not contain the identities of all

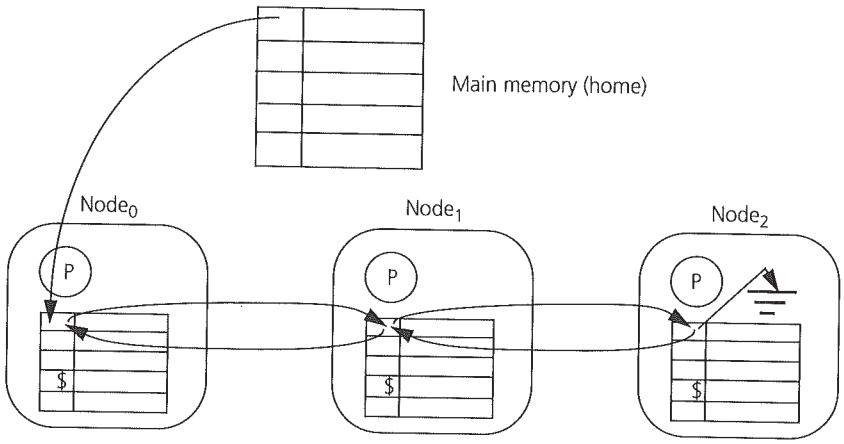


FIGURE 8.8 A doubly linked-list distributed directory organization. A cache line contains not only data and state for the block but also forward and backward pointers for the distributed linked list.

sharers but only a pointer to the first sharer in the list plus a few state bits. This pointer is called the *head pointer* for the block. The remaining nodes caching that block are joined together (using additional pointers that are associated with each cache line in a node) in a *distributed, doubly linked list*—that is, a cache that contains a copy of the block also contains pointers to the next and previous caches that have a copy, called the *forward* and *backward* pointers, respectively (see Figure 8.8).

On a read miss, the requesting node sends a network transaction to the home memory to find out the identity of the head node of the linked list, if any, for that block. If the head pointer is null (no current sharers), the home replies with the data. If the head pointer is not null, then the requestor must be added to the list of sharers. The home responds to the requestor with the head pointer. The requestor then sends a message to the head node, asking to be inserted at the head of the list and hence to become the new head node. The net effect is that the head pointer at the home now points to the requestor, the forward pointer of the requestor's own cache entry points to the old head node (which is now the second node in the linked list), and the backward pointer of the old head node points to the requestor. The data for the block is provided by the home if it has the latest copy or by the head node, which always has the latest copy (is the owner) otherwise.

On a write miss, the writer again obtains the identity of the head node, if any, from the home. It then inserts itself into the list as the head node as before (if the writer was already in the list as a sharer and is now performing an upgrade, it is deleted from its current position in the list and inserted as the new head). Following this, the rest of the distributed linked list is traversed node by node via network transactions to find and invalidate successive copies of the block. If a block that is written is shared by three nodes A, B, and C, the home only knows about A so the writer sends an invalidation message to it; the identity of the next sharer B can only

be known once A is reached, and so on. Acknowledgments for these invalidations are sent to the writer. Once again, if the data for the block is needed by the writer, it is provided by either the home or the head node as appropriate. The number of messages per invalidating write—the bandwidth demand—is proportional to the number of sharers as in the memory-based schemes, but now so is the number of messages in the critical path, that is, the latency. Each of these serialized messages invokes the communication assist at its destination, increasing latency and overall assist occupancy further. In fact, even a read miss to a clean block involves the assists of three nodes to insert the node in the linked list.

Write backs or other replacements from the cache also require that the node delete itself from the sharing list, which means communicating with the nodes that are before and after it in the list. This is necessary because the new block that replaces the old one in the cache will need the forward and backward pointer slots of the cache entry for its own sharing list. Synchronization is required to avoid simultaneous replacement of adjacent nodes in the list, and the involvement of multiple nodes increases overall assist occupancy. An example cache-based protocol is described in more depth in Section 8.6.

To counter the latency and occupancy disadvantages, cache-based schemes have some important advantages over memory-based schemes. First, the directory overhead is small. Every block in main memory has only a single head pointer. The number of forward and backward pointers is proportional to the number of cache blocks in the machine, which is much smaller than the number of memory blocks. The second advantage is that a linked list records the order in which accesses were made to memory for the block, thus making it easier to provide fairness and to avoid livelock in a protocol (most memory-based schemes do not keep track of request order, as we will see). Third, the work to be done by assists in sending invalidations is not centralized at the home but rather distributed among sharers, thus perhaps spreading out assist occupancy and reducing the corresponding bandwidth demands placed on a particularly busy home assist.

Manipulating insertion in and deletion from distributed linked lists can lead to complex protocol implementations. For example, deleting a node from a sharing list requires careful coordination and mutual exclusion with processors ahead of and behind it in the linked list since those processors may also be trying to replace the same block concurrently. These complexity issues have been greatly alleviated by the formalization and publication of a standard for a cache-based directory organization and protocol: the IEEE 1596-1992 Scalable Coherent Interface (SCI) standard (Gustavson 1992). The standard includes a full specification and C code for the protocol. Several commercial machines use this protocol (e.g., Sequent NUMA-Q [Lovett and Clapp 1996], Convex Exemplar [Convex Computer Corporation 1993; Thekkath et al. 1997], and Data General [Clark and Alnes 1996]), and variants that use alternative list representations (singly linked lists instead of the doubly linked lists in SCI) have also been explored (Thapar and Delagi 1990). We shall examine the SCI protocol itself in more detail in Section 8.6 and so defer detailed discussion of advantages and disadvantages.

Summary of Directory Organization Alternatives

To summarize, there are many different ways to organize how directories store the cache state of memory blocks. Simple bit vector representations work well for machines that have a moderate number of nodes visible to the directory protocol. For larger machines, many alternatives are available to reduce the memory overhead. The organization chosen does, however, affect the complexity of the coherence protocol and the performance of the directory scheme against various sharing patterns. Hierarchical directories have not been popular on real machines, whereas machines with flat memory-based and cache-based (linked-list) directories have been built and used for some years now.

The next section quantitatively assesses the behavior of parallel programs and the implications for directory-based approaches as well as some important protocol and architectural trade-offs at this basic level.

8.3 ASSESSING DIRECTORY PROTOCOLS AND TRADE-OFFS

As in Chapter 5, this section uses a simulator to examine some relevant characteristics of applications that can inform architectural trade-offs but that cannot be measured on real machines. Issues such as three-state versus four-state or invalidation-based versus update protocols that were discussed in Chapter 5 are not revisited here. The focus is on invalidation-based protocols, since update protocols have an additional disadvantage in scalable machines: useless updates incur a separate network transaction for each destination rather than a single bus transaction that is snooped by all caches. In addition, update-based protocols make it much more difficult to preserve the desired memory consistency model in directory-based systems. This section quantifies the distribution of invalidation patterns in directory protocols, examines how the distribution of traffic between local and remote changes as the number of processors is increased for a fixed problem size, and revisits the impact of cache block size on traffic. In all cases, the experiments assume a memory-based flat directory protocol. Two changes are made from the experiments in Chapter 5. Since Radix sorting would exhibit a lot of false sharing at larger processor counts (our default here is 32 rather than 16 processors), we use a problem size of 1M rather than 256K keys. And we use 8-KB rather than 64-KB caches in all our smaller cache size experiments, to see the effect of even fewer working sets fitting in the cache.

8.3.1 Data Sharing Patterns for Directory Schemes

It was claimed earlier that the number of invalidations that need to be sent out on a write is usually small, which makes directories especially valuable and can reduce directory storage overhead without hurting performance. This subsection quantifies that claim for our parallel application case studies. It also develops a framework for categorizing data structures in terms of sharing patterns and understanding how the invalidation patterns scale, and explains the behavior of the application case studies in light of this framework. The simulated protocol assumes only the three basic cache states (MSI) for simplicity.

Sharing Patterns for Application Case Studies

For invalidation-based directory protocols, it is important to understand two aspects of an application's data sharing patterns: (1) the frequency with which processors issue writes that may require invalidating other copies (i.e., writes to data that is not in modified state in the writer's cache in an MSI protocol, or *invalidating writes*), called the *invalidation frequency*; and (2) the distribution of the number of invalidations (sharers) needed upon these writes, called the *invalidation size distribution*. Directory schemes are particularly advantageous if the average invalidation size is small and the frequency is significant enough that using broadcast all the time would indeed be a performance problem. Figure 8.9 shows the invalidation size distributions for our parallel application case studies running on 64-node systems (one processor per node) for the default problem sizes presented in Chapter 4. Infinite per-processor caches are used in these simulations to capture inherent sharing patterns. With finite caches, replacement hints sent to the directory may turn off presence bits and reduce the number of invalidations sent on writes in some cases (though traffic will not be reduced since the replacement hints have to be sent). A write may send zero invalidations in an MSI protocol if the block was loaded in shared state but there are currently no other sharers. This would not happen with infinite caches in a MESI protocol. With infinite caches, invalidation frequency is proportional to the communication-to-computation ratio.

It is clear that the invalidation sizes are usually small, indicating both that directories are indeed likely to be very useful in containing traffic and that it is not necessary for the directory to maintain a presence bit per processor in a flat memory-based scheme. The nonzero frequencies of very large invalidation sizes are usually due to synchronization variables, where many processors spin on a variable and one processor writes it, invalidating them all. We are interested not just in the results for a given problem size and number of processors but also in how they scale. The communication-to-computation ratios discussed in Chapter 4 give us a good idea about how the frequency of invalidating writes should scale. For the size distributions, we can appeal to our understanding of applications and their usage of data structures (and validate with experiments), which can also help explain the basic results observed in Figure 8.9.

A Framework for Sharing Patterns

Data access patterns in applications can be categorized in many ways: predictable versus unpredictable, regular versus irregular, coarse-grained versus fine-grained (or contiguous versus noncontiguous in the address space), near-neighbor versus long-range in an interconnection topology, and so on. For understanding invalidation patterns, the relevant categories are read-only, producer-consumer, migratory, and irregular read-write. (A similar categorization can be found in [Gupta and Weber 1992].)

- *Read-only.* Read-only data structures are never written once they have been initialized. There are no invalidating writes, so data in this category is not an

issue for directories. Examples include program code and the scene data in the Raytrace application.

- *Producer-consumer.* A processor produces (writes) a data item, then one or more processors consume (read) it, then a processor produces it again, and so on. Flag-based synchronization is an example, as is the near-neighbor sharing in an iterative grid computation. The producer may be the same process every time or it may change; for example, in a branch-and-bound algorithm, the bound variable may be written by different processes as they find improved bounds. The invalidation size for this category is determined by how many consumers there have been each time the producer writes the value. We can have situations with one consumer, all processes being consumers, or a few processes being consumers. These situations may have different frequencies and scaling properties, although for most applications either the size does not scale quickly with the number of processors or the frequency has been found to be low.[1]
- *Migratory.* Migratory data bounces around, or migrates, from one processor to another, being written (and usually read) by each processor to which it bounces. An example is a global sum, into which different processes add their partial sums. Each time a processor writes the variable, only the previous writer has a copy (since it invalidated the previous "owner" when it did its write); so only a single invalidation is generated upon a write, regardless of the number of processors used.
- *Irregular read-write.* This corresponds to irregular or unpredictable read and write access patterns to data by different processes. A simple example is a distributed task-queue system. Processes will probe (read) the head pointer of a task queue when they are looking for work to steal, and this head pointer will be written when a task is added at the head. These and other irregular patterns usually lead to wide-ranging invalidation size distributions, but in most observed applications the frequency concentration tends to be very much toward the small end of the spectrum (see the Radiosity example in Figure 8.9).

1. Examples of the producer-consumer size distribution not scaling up are the noncorner border elements in a near-neighbor regular grid partition and the key permutations in Radix. They lead to an invalidation size of one, which does not increase with the number of processors or the problem size. Examples of all processes being consumers (invalidation size $p - 1$) are a global energy variable that is read by all processes during a time-step of a physical simulation and then written by one at the end or a synchronization variable on which all processes spin. While the invalidation size here is large, such writes fortunately tend to happen very infrequently in real applications. Finally, examples of a few processes being consumers are the corner elements of a grid partition or the flags used for tree-based synchronization. This leads to an invalidation size of a few, which may or may not scale with the number of processors (it doesn't in these two examples).

(continued)

FIGURE 8.9 Invalidation patterns with default data sets and 64 processors

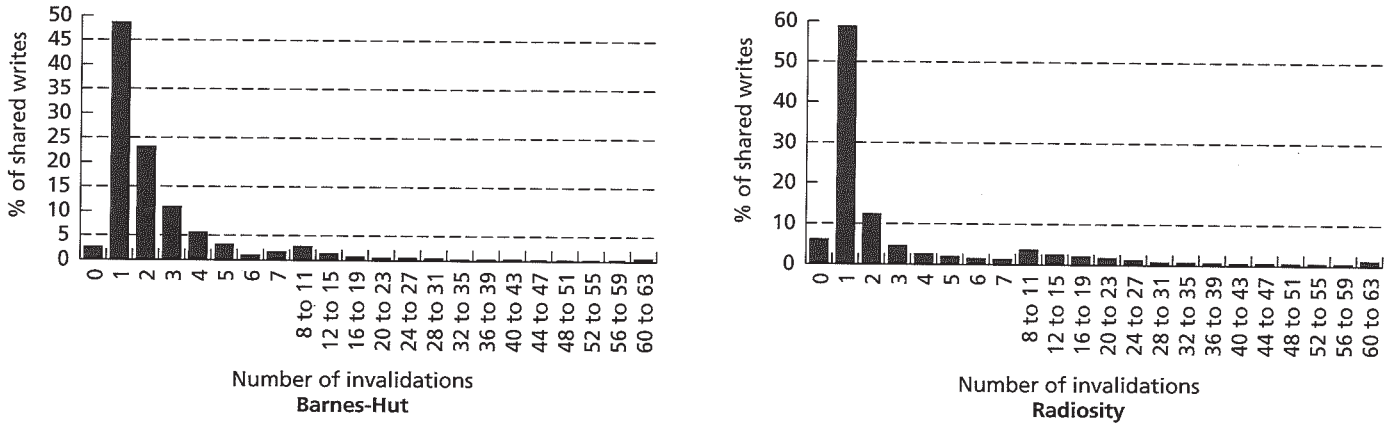


FIGURE 8.9 Invalidation patterns with default data sets and 64 processors. The *x*-axis shows the invalidation size (number of active sharers) upon an invalidating write, and the *y*-axis shows the percentage of invalidating writes whose size takes that *x*-axis value. The invalidation size distribution was measured by simulating a full bit vector directory representation and recording for every invalidating write how many presence bits (other than the writer's) are set when the write occurs. The data is averaged over all the 64 processors used.

Applying the Framework to the Application Case Studies

Let us now look briefly at each of the applications in Figure 8.9 to interpret the results in light of these four sharing patterns and to understand how the size distributions might scale.

In the LU factorization program, when a block is written it has previously been read only by the same processor that is doing the writing (the process to which it is assigned). This means that no other processor should have a cached copy, and zero invalidations should be sent. Once it is written, it is read by several other processes and no longer written further. The reason that we see one invalidation being sent in the figure is that the matrix is initialized by a single process; particularly with the infinite caches we use, that process has a copy of the entire matrix in its cache and will be invalidated the first time another processor does a write to a block. An insignificant number of invalidating writes invalidates all processes, which is due to some global variables and not the main matrix data structure. Scaling the problem size or the number of processors does not change the invalidation size distribution for the matrix but only for the global variables. Of course, the invalidation frequencies do change with scaling, just like the communication-to-computation ratio.

In the Radix sorting kernel, invalidations are sent in two producer-consumer situations. In the permutation phase, the word or block written has been read since the last write only by the process to which that key is assigned, so at most a single invalidation is sent out. The same key position in the destination array may be written by different processes in different outer loop iterations of the sort; however, in each iteration there is only one reader of a key, so even this infrequent case generates only two invalidations (one to the reader and one to the previous writer). If there is false sharing, all sharers are writing the block, so there is only one invalidation each time. The other situation that generates invalidations is the histogram accumulation, which is done in a tree-structured fashion and usually leads to a small number of invalidations at a time. These invalidations to multiple sharers are clearly very infrequent. In Radix too, increasing the problem size does not change the invalidation size in either phase (though it may change the relative invalidation frequencies in the two phases), whereas increasing the number of processors increases the sizes but only in some infrequent parts of the histogram accumulation phase. The dominant pattern by far remains 0 or 1 invalidations.

The nearest-neighbor, producer-consumer communication pattern on a regular grid in Ocean leads to most of the invalidations being sent to 0 or 1 processes (at the borders of a partition). At partition corners, more frequently encountered in the multigrid equation solver, two or three sharers may need to be invalidated. This does not grow with problem size or number of processors. At the highest levels of the multigrid hierarchy, the border elements of a few processors' partitions might fall on the same cache block, causing four or five sharers to be invalidated. There are also some global accumulator variables, which display a migratory sharing pattern (1 invalidation), and a couple of very infrequently used one-producer, all-consumer global variables (other than synchronization variables).

In Raytrace the dominant data structure is the scene data, which is read-only. The major read-write data consists of the image and the task queues. Each word in the image is written only once by one processor per frame. This leads to either 0 invalidations if the same processor writes a given image pixel in consecutive frames (as is usually the case) or 1 invalidation if different processors do, as might be the case when tasks are stolen or if there is write-write false sharing. The task queues themselves lead to the irregular read-write access patterns discussed earlier, with a wide-ranging distribution that is dominated in frequency by the low end (hence the very small nonzeros all along the *x*-axis in this case). Here too we find some infrequently written one-producer, all-consumer global variables.

In the Barnes-Hut application, the important data is the body and cell positions, the pointers used to link up the tree, and some global variables used as energy values. The position data is of the producer-consumer type. A given body's position is usually read by one or a few processors during the force calculation (tree traversal) phase. The positions (centers of mass) of the cells are read by many processes, the number increasing toward the root, which is read by all. This data thus causes a fairly wide range of invalidation sizes when it is written by its assigned processor in the update and tree construction phases that follow force calculation. The root and upper-level cells are responsible for invalidations being sent to all processors, but their frequency is quite small. The tree pointers are similar in their behavior to the cell centers of mass. The first write to a pointer in the tree-building phase invalidates the caches of the processors that read it in the previous force calculation phase; subsequent writes invalidate those processors that have read the pointer during the current tree-building phase, which is an irregularly sized but mostly small set of processors. As the number of processors is increased, the invalidation size distribution tends to shift to the right as more processors tend to read a given item, but the shift is slow and the dominant invalidations are still to a small number of processors. The reverse effect (also slow) is observed when the number of bodies is increased.

Finally, the Radiosity application has very irregular access patterns to many different types of data, including data that describes the scene (patches and elements) and the task queues. The resulting invalidation patterns show a wide distribution; however, even here the greatest frequency by far is concentrated toward 0 to 2 invalidations. Many of the accesses to the scene data behave in a migratory way, as do a few counters, and a couple of global variables are one-producer, all-consumer.

The empirical data and categorization framework indicate that in most cases the invalidation size distribution is dominated by small numbers of invalidations. The common use of parallel machines as multiprogrammed compute servers for sequential or small-way parallel applications further limits the number of sharers (process migration usually leads to invalidations of size 1). Sharing patterns that cause large numbers of invalidations are empirically found to be very infrequent at run time. A possible exception is highly contended synchronization variables, which are usually handled specially by software or hardware, as we shall see. In addition to validating the directory-based approach and suggesting its potential for performance scalability, these results suggest that limited-pointer directory representations should be successful since the frequency of overflows will be small.

8.3.2 Local versus Remote Traffic

A key property for systems with distributed memory is how much of the traffic due to cache misses or protocol activity is kept within a node (local) rather than going on the interconnect (remote). For a given number of processors and machine organization, the fraction of traffic that is local depends on the problem size. However, it is instructive to examine how the traffic and its distribution change with the number of processors even when the problem size is held fixed (i.e., under PC scaling). Figure 8.10 shows the results for the default problem sizes, breaking down the remote traffic into various categories such as sharing (true or false), capacity, cold start, write back, and overhead. A MESI rather than MSI protocol is used in this and the next subsection. Overhead includes the fixed header sent across the network with each cache block of data as well as the traffic associated with protocol transactions like invalidations and acknowledgments that do not carry any data. This protocol traffic component is different from that on a bus-based machine: each individual point-to-point invalidation consumes traffic, and acknowledgments place traffic on the interconnect too. Traffic is shown in bytes per FLOP or bytes per instruction for different applications.

We can see that both local traffic and capacity-related remote traffic tend to decrease when the number of processors increases, due to both decrease in per-processor working sets and decrease in cold misses that are satisfied locally instead of remotely. However, sharing-related traffic increases as expected. In applications with small working sets, like Barnes-Hut, LU, and Radiosity, the fraction of capacity-related traffic is very small, at least beyond a couple of processors. In irregular applications like Barnes-Hut and Raytrace, most of the capacity-related traffic is remote, all the more so as the number of processors increases, since data cannot be distributed easily at page granularity for capacity misses to be satisfied locally. In cases like Ocean, the capacity-related traffic is substantial even with the large cache and is almost entirely local when pages are placed properly (which can be done quite easily with 4D array data structures). With round-robin placement of shared pages, we would have seen most of the local capacity misses in Ocean turn to remote ones.

When we use smaller caches to capture the realistic scenario of working sets not fitting in the cache in Ocean and Raytrace, capacity traffic becomes much larger. In Ocean, most of this traffic is still local due to good data distribution, and the trend for remote traffic versus number of processors doesn't change. Poor distribution of pages would have swamped the network with traffic, but with proper distribution, remote traffic is quite low. In Raytrace, however, the capacity-related traffic is mostly remote, and the fact that it now dominates changes the slope of the curve of total remote traffic compared to that with large caches, where sharing traffic dominates. Remote traffic still increases with the number of processors but much more slowly since the working set size and, hence, capacity miss rate does not depend as much on the number of processors as the sharing miss rate.

When a miss is satisfied remotely, whether it is satisfied at the home or it needs another message to obtain the data from a dirty node depends not only on whether it is a sharing miss or a capacity/conflict/cold miss but also on the size of the cache. In

a small cache, dirty data may be replaced and written back, so a sharing miss by another processor may be satisfied at the home node rather than at the previously dirty node. For applications like Ocean that allow data to be placed easily in the memory of the node to which they are assigned (i.e., to be appropriately distributed for locality), it is often the case that only that node writes the data, so even if the data is found dirty, it is found so in a cache at the home node itself. The extent to which this is true depends on the data access patterns of the application, the granularity of data allocation in memory, and whether the data is indeed distributed properly by the program.

8.3.3 Cache Block Size Effects

The effects of block size on cache miss rates and bus traffic were assessed in Chapter 5, at least up to 16 processors. For miss rates, the trends beyond 16 processors extend quite naturally, except for threshold effects in the interaction of problem size, number of processors, and block size, as discussed in Chapter 4. This section examines the impact of block size on the components of local and remote traffic in machines with distributed memory.

Figure 8.11 shows how traffic scales with block size for 32-processor executions of the applications with 1-MB caches per processor. In Barnes-Hut, the overall traffic increases slowly until about a 64-byte block size and more rapidly thereafter primarily due to false sharing. However, the amount of traffic is small. Since the overhead per block moved through the network is fixed (as is the cost of invalidations and acknowledgments), the overhead component tends to shrink with increasing block size to the extent that there is spatial locality (i.e., if larger blocks reduce the number of blocks transferred). LU has perfect spatial locality, so the data traffic remains fixed as block size increases. Overhead is reduced, so overall traffic in fact shrinks with increasing block size. In Raytrace, the remote capacity traffic has poor spatial locality, so it grows quickly with block size. In both Barnes-Hut and Raytrace, the true sharing traffic has poor spatial locality too, as is the case in Ocean at column-oriented partition borders (spatial locality even on remote data is good at row-oriented borders). Finally, the graph for Radix clearly shows the impact of false sharing on remote traffic when it occurs past the threshold block size (here about 128 or 256 bytes). Results with smaller caches show the behavior of capacity misses playing a dominant role, as expected.

8.4 DESIGN CHALLENGES FOR DIRECTORY PROTOCOLS

Designing a correct, efficient directory protocol involves issues that are more complex and subtle than the simple organizational choices we have discussed so far, just as designing a bus-based protocol was more complex than choosing the number of states and drawing the state transition diagram for stable states. We had to deal with the nonatomicity of state transitions, split-transaction buses, serialization and ordering issues, deadlock, livelock, and starvation. Now that we understand the basics of

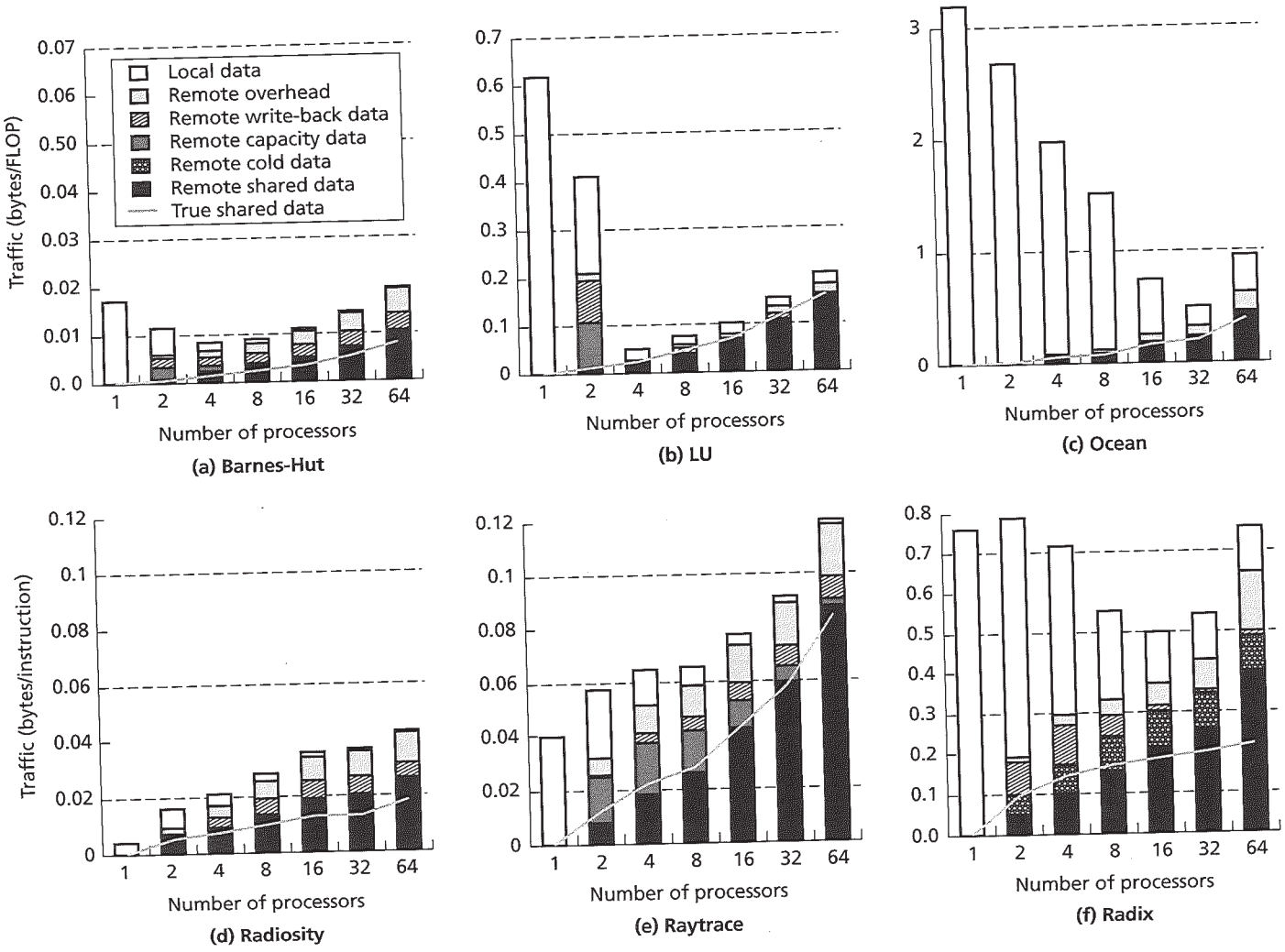


FIGURE 8.10 Traffic versus number of processors

(continued)

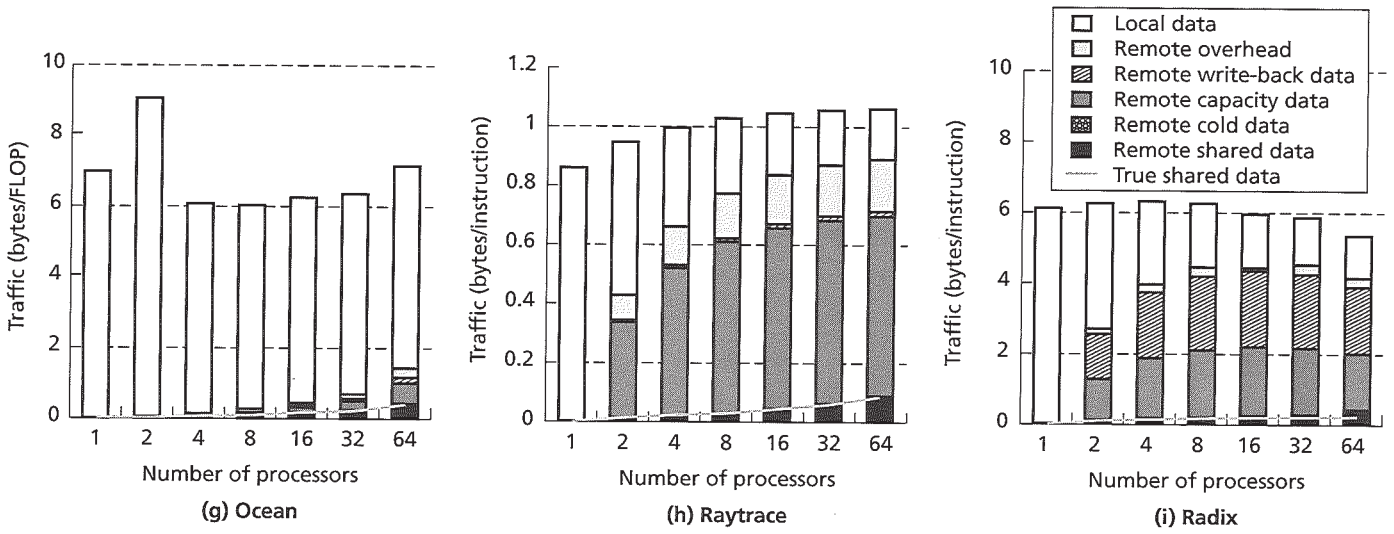


FIGURE 8.10 Traffic versus number of processors. The overhead per block transferred across the network is 8 bytes. No overhead is considered for local accesses. Graphs (a)–(f) assume 1-MB caches per processor and graphs (g)–(i), 8 KB. The caches have a 64-byte block size and are four-way set associative.

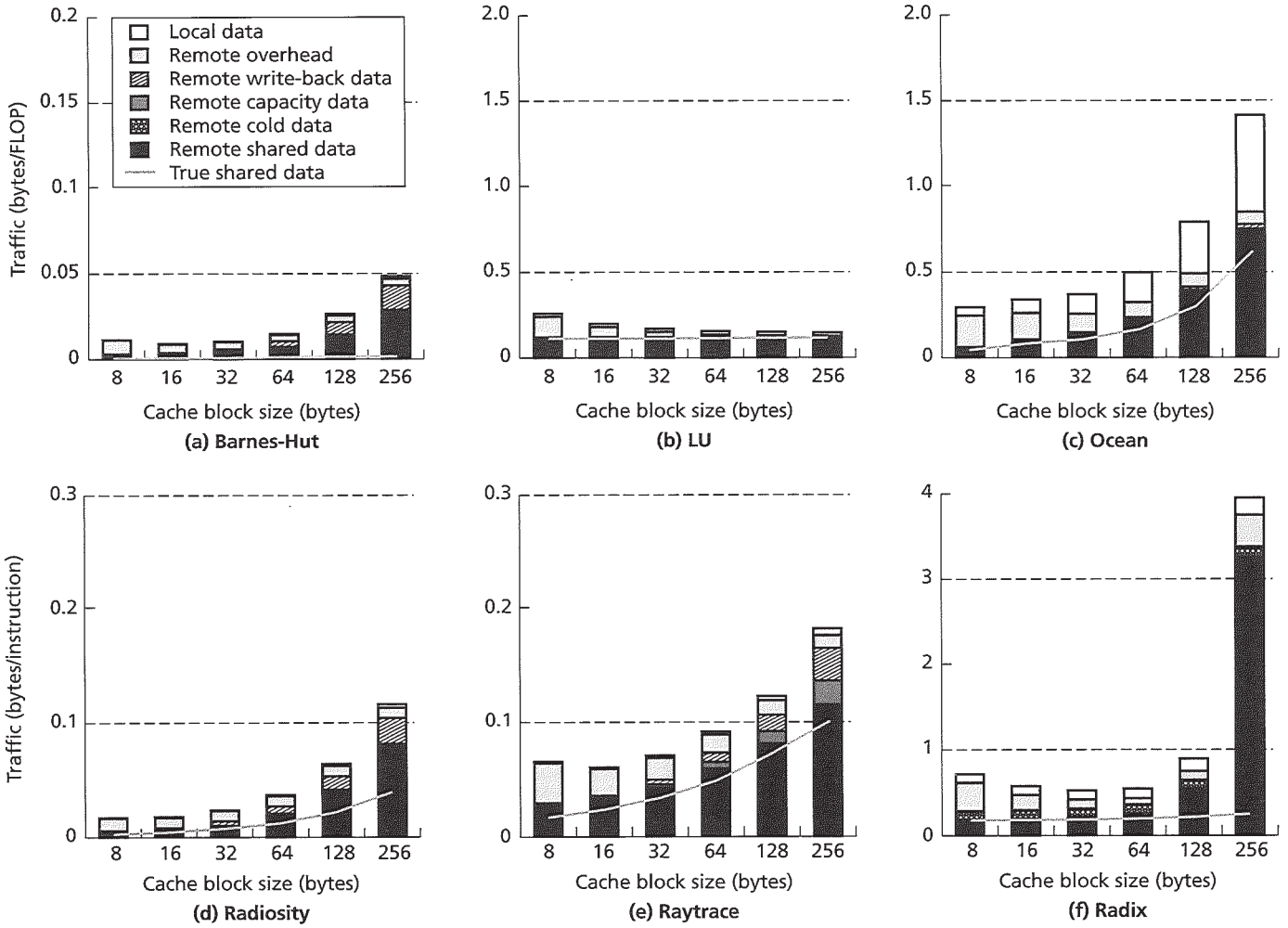


FIGURE 8.11 Traffic versus cache block size

(continued)

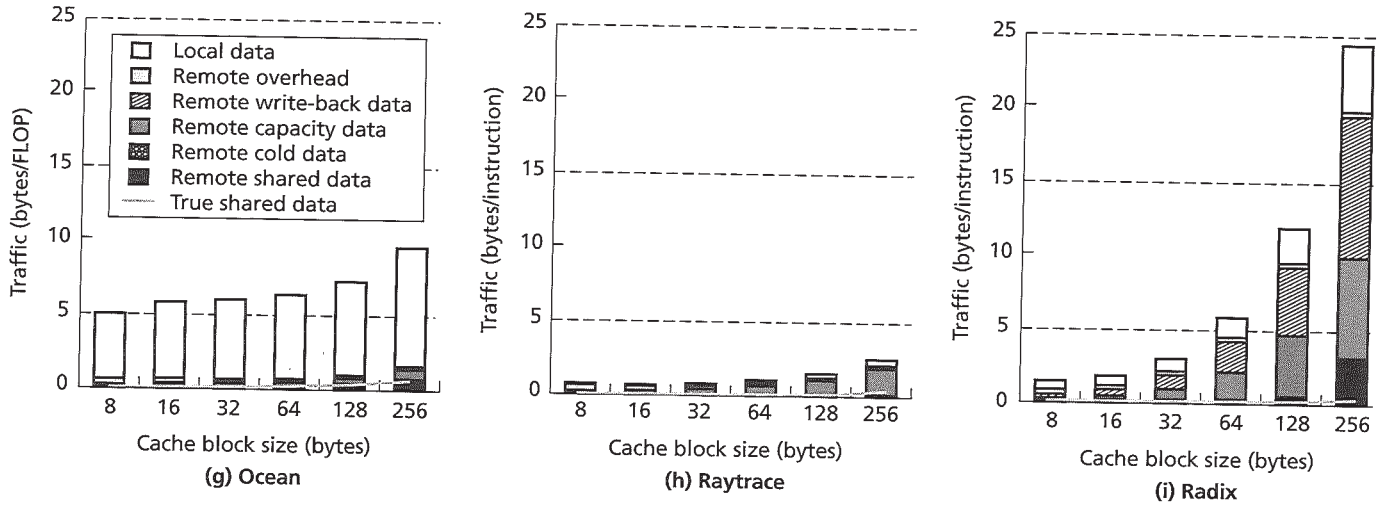


FIGURE 8.11 Traffic versus cache block size. Data are shown for 32-processor executions. The overhead per block transferred over the network is 8 bytes. No overhead is considered for local access. Graphs (a)–(f) show the data for 1-MB caches per processor and (g)–(i) for 8-KB caches. All caches are four-way set associative.

directories, we are ready to dive into these issues for them as well. This section discusses the new protocol-level design challenges that arise in correctly implementing directory protocols for high performance and identifies general techniques for addressing these challenges. In the next two sections, the techniques are specialized in the case studies of memory-based and cache-based directory protocols.

As always, the design challenges for scalable coherence protocols are to provide high performance while preserving correctness and to contain the complexity that results. Let us look at performance and correctness in turn, focusing on issues that were not already addressed for bus-based or noncaching systems. Since performance optimizations tend to increase concurrency and complicate correctness, let us examine them first.

8.4.1 Performance

The network transactions on which cache coherence protocols are built differ from those used in explicit message passing in two ways. First, they are automatically generated by the system—in particular, by the communication assists or controllers—in accordance with the protocol. Second, they are individually small, each carrying either a request, an acknowledgment, or a cache block of data plus some control bits. However, the basic performance model for network transactions developed in earlier chapters applies here as well. A typical network transaction incurs some overhead on the processor at its source (traversing the cache hierarchy on the way out and back in); some work or occupancy on the communication assists at its endpoints (typically looking up state, generating requests, or intervening in the cache); and some delay in the network due to transit latency, network bandwidth, and contention. Typically, the processor itself is not directly involved at the home, the dirty node, or the sharers but only at the requestor (although it may suffer at the other nodes as well due to contention).

It is useful to understand performance in terms of the layers of a multiprocessor system introduced earlier (Figure 8.2). The protocol layer of a system implements the programming model, using the network transactions provided by the communication abstraction. Thus, the protocol layer does not have much leverage on the basic communication costs of a single network transaction—transit latency, network bandwidth, assist occupancy, and processor overhead—but it can determine the number and structure of the network transactions needed to realize memory operations like reads and writes under different circumstances. In general, there are three classes of techniques for improving performance: (1) protocol optimizations, (2) high-level machine organization, and (3) hardware specialization to improve the basic communication parameters. The first two assume a fixed set of performance parameters for the communication architecture and are discussed in this section. The impact of varying the basic performance parameters will be examined in Section 8.7.

Protocol Optimizations

The two major performance goals at the protocol level are to reduce the number of network transactions generated per memory operation, which reduces the bandwidth demands placed on the network and the communication assists; and to reduce the number of actions, especially network transactions, that are on the critical path of the processor, thus reducing uncontended latency. The latter can be done by overlapping the transactions needed for a memory operation as much as possible. To some extent, protocol design can also help reduce the endpoint assist occupancy per transaction—especially when the assists are programmable—which reduces both uncontended latency as well as endpoint contention. The traffic, latency, and occupancy characteristics should not scale up quickly with the number of processing nodes used and should perform gracefully under pathological conditions like hot spots.

As we have seen, the manner in which directory information is stored determines the number of network transactions in the critical path of a memory operation. For example, a memory-based protocol can issue invalidations in an overlapped manner from the home whereas, in a cache-based protocol, the distributed list must be walked by network transactions to learn the identities of the sharers. However, even within a class of protocols, there are many ways to improve performance.

Consider a read miss to a remotely allocated block that is dirty in a third node in a flat, memory-based protocol. The strict request-response option described earlier is shown in Figure 8.12(a). The home responds to the requestor with a message containing the identity of the owner node. The requestor then sends a request to the owner, which replies to it with the data (the owner also sends a "revision" message to the home, which updates memory with the data and sets the directory state to be shared).

There are four network transactions in the critical path for the read operation and five transactions in all. One way to reduce these numbers is *intervention forwarding*. In this case, the home does not respond to the requestor but simply forwards the request as an intervention transaction to the owner, asking it to retrieve the block from its cache. An intervention is just like a request but is issued in reaction to a request and is directed at a cache rather than memory (it is similar to an invalidation in this sense but also seeks data from the cache). The owner then replies to the home with the data or an acknowledgment (if the block is in exclusive rather than modified state), at which time the home updates its directory state and replies to the requestor with the data (Figure 8.12[b]). Intervention forwarding reduces the total number of transactions needed to four, reducing bandwidth needs, but all four are still in the critical path. A more aggressive method is *reply forwarding* (Figure 8.12[c]). Here too, the home forwards the intervention message to the owner node, but the intervention contains the identity of the requestor and the owner replies directly to the requestor itself. The owner also sends a revision message to the home so that the memory and directory can be updated, but this message is not in the critical path of the read miss. This keeps the number of transactions at four but reduces

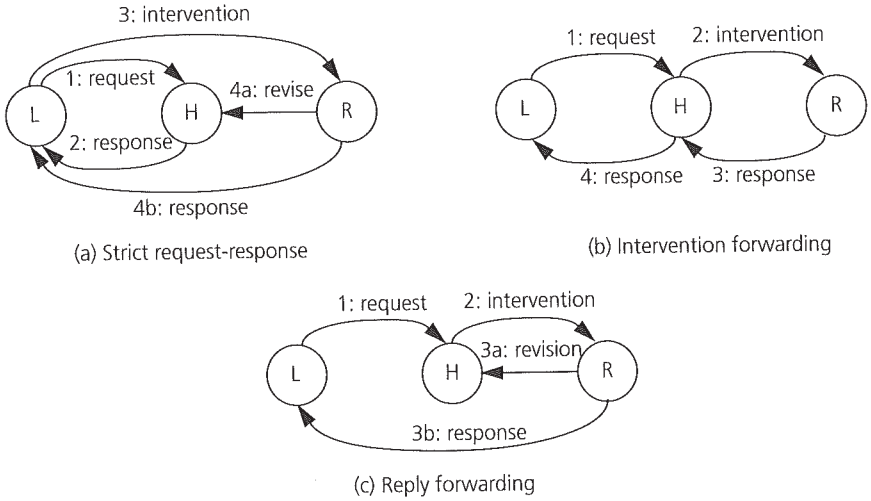


FIGURE 8.12 Reducing latency in a flat, memory-based protocol through forwarding. The case shown is of a read request to a block in exclusive state. L represents the local or requesting node, H is the home for the block, and R is the remote owner node that has the exclusive copy of the block.

the number in the critical path to three (request → intervention → reply-to-requestor); it is, therefore, called a *three-message miss*. Notice that with either of intervention forwarding or reply forwarding the protocol is no longer strictly request-response since a request to the home generates another request (to the owner node, which in turn generates a response). This can complicate deadlock avoidance, as we shall see later.

Besides being only intermediate in its latency and traffic characteristics, intervention forwarding has the disadvantage that outstanding intervention requests are kept track of at the home rather than at the requestor, since responses to the interventions are sent to the home. Because requests that cause interventions may come from any of the nodes, the home node must keep track of up to $k*P$ interventions at a time, where k is the number of outstanding requests allowed per node. A requestor, on the other hand, would only have to keep track of at most k outstanding interventions. Reply forwarding does not require the home to keep track of outstanding requests and also has better performance characteristics, so systems prefer to use it. Similar forwarding techniques can be used to reduce latency in cache-based schemes at the cost of strict request-response simplicity, as shown in Figure 8.13.

In addition to forwarding, other protocol optimizations to reduce latency include overlapping transactions and activities by performing them speculatively. For example, when a request arrives at the home, the assist can read the data from memory in parallel with the directory lookup, in the hope that in most cases the block will indeed be clean at the home. If the directory lookup indicates that the block is dirty in some cache, then the memory access is wasted and must be ignored. Finally, pro-

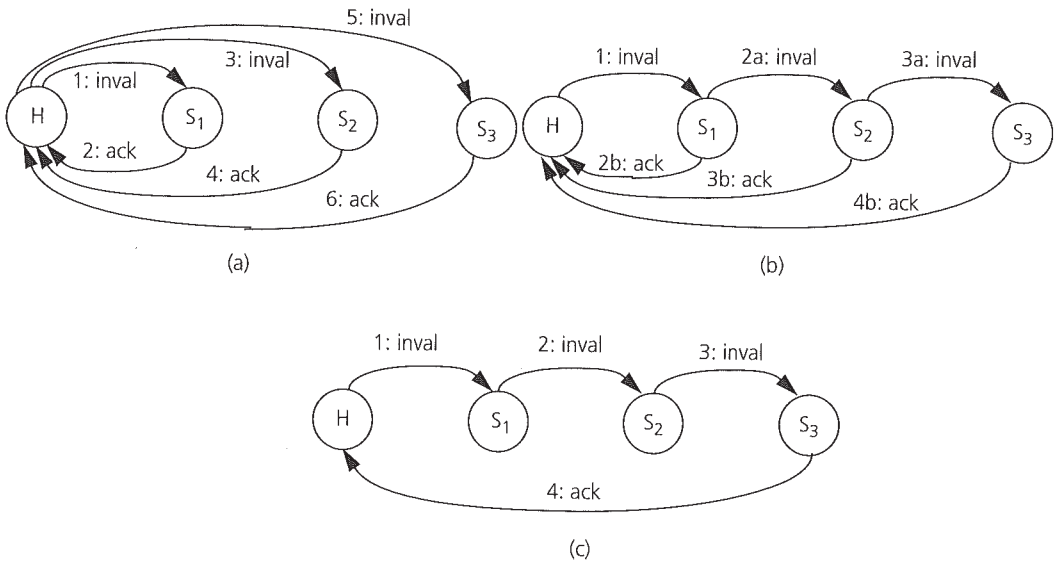


FIGURE 8.13 Reducing latency in a flat, cache-based protocol. In this scenario, invalidations are sent from the home H to the sharers S_i on a write operation. In the strict request-response case (a), every node includes in its acknowledgment (response) the identity of the next sharer on the list, and the home then sends that sharer an invalidation. The total number of transactions in the invalidation sequence is $2s$, where s is the number of sharers and all are in the critical path. In (b), each invalidated node forwards the invalidation to the next sharer and in parallel sends an acknowledgment to the home. The total number of transactions is still $2s$, but only $s + 1$ are in the critical path. In (c), only the last sharer on the list sends a single acknowledgment telling the home that the sequence is done. The total number of transactions is $s + 1$. (b) and (c) are not strict request-response cases.

tocols may also automatically detect common sharing patterns to which the standard invalidation-based protocol is not ideally suited and adjust themselves at run time to interact better with these patterns (see Exercises 8.9 and 8.10).

High-Level Machine Organization

Machine organization can interact with the protocol to help improve performance as well. For example, the use of large tertiary caches within a node can reduce the number of protocol transactions generated by artifactual communication. For a fixed total number of processors, using multiprocessor rather than uniprocessor nodes in a two-level organization may be useful as well.

The potential advantages of a two-level organization are in both cost and performance. On the cost side, certain fixed per-node costs may be amortized among the processors within a node, and it is possible to use existing SMPs that may themselves be commodity parts. On the performance side, advantages may arise from sharing characteristics that reduce the number of accesses that involve the directory protocol and generate network transactions across nodes. If one processor brings a

block of data into its cache, another processor in the same node may be able to satisfy its miss to that block (for the same or a different word) more quickly through the local protocol using cache-to-cache sharing, especially if the block is allocated remotely. Requests may also be combined: if one processor has a request outstanding to the directory protocol for a block, another processor's request within the same SMP can be combined with, and obtain the data from, the first processor's response, reducing latency, network traffic, and potential hot spot contention. These advantages are similar to those of full hierarchical approaches and of shared caches. In fact, within an SMP, processors may even share a cache at some level of the hierarchy, in which case all the trade-offs for shared caches discussed in Chapter 6 apply. With fewer nodes, more of the main memory is local as well. Finally, cost and performance characteristics may be improved by using a hierarchy of packaging technologies appropriately.

Of course, the extent to which the two-level sharing hierarchy can be exploited depends on the locality in the sharing and data access patterns of applications, how well processes are mapped to processors in the hierarchy, and the cost difference between communicating within a node and across nodes. For example, applications that have wide but physically localized read-only sharing in a phase of computation, like the Barnes-Hut galaxy simulation, can benefit significantly from cache-to-cache sharing if the miss rates are high to begin with. Applications that exhibit nearest-neighbor sharing (like Ocean) can also have most of their accesses satisfied within a multiprocessor node if processes are mapped properly to nodes. However, although some processes may have all their accesses satisfied within their node, others will have accesses along at least one border satisfied remotely, so load imbalances will result and the benefits of the hierarchy will be diminished (performance will be limited by that of the most penalized processor). In all-to-all communication patterns, the savings in inherent communication is more modest. Instead of communicating with $p - 1$ remote processors in a p-processor system, a processor now communicates with $k - 1$ local processors and $p - k$ remote ones (where k is the number of processors within a node), a savings of at most $(p - k)/(p - 1)$ in internode communication. Finally, with several processes sharing a main memory unit, it may also be easier to distribute data appropriately among processors at page granularity. Some of these trade-offs and application characteristics are explored quantitatively in (Weber 1993; Erlichson et al. 1995). Of our two case study machines, the Sequent NUMA-Q uses four-processor, bus-based, cache-coherent SMPs as the nodes. The SGI Origin takes an interesting position: two processors share a bus and memory (and a board) to amortize cost, but they are not kept coherent by a snoopy protocol on the bus; rather, a single directory protocol keeps all caches in the machine coherent.

Compared to using uniprocessor nodes, the major potential disadvantage of using multiprocessor nodes is the sharing of communication resources by processors within a node. When processors share a bus, an assist, or a network interface, they amortize its cost but compete for its bandwidth. If their bandwidth demands are not reduced much by locality in sharing patterns, the resulting contention can hurt performance. The solution is to increase the throughput of these resources as well when processors are added to the node, but this compromises the cost advantages. Sharing

a bus within a node has some particular disadvantages. First, if the bus has to accommodate several processors, it becomes longer and is not likely to be contained in a single board or other packaging unit. These effects slow the bus down, increasing the latency to both local and remote data. Second, if the bus supports snooping coherence within the node, a request that must be satisfied remotely typically has to wait for local snoop results to be reported before it is sent out to the network, causing unnecessary delays. Third, with a snooping bus at the remote node too, many references that do go remote will require snoops and data transfers on the local bus as well as the remote bus, increasing latency and reducing effective data access bandwidth. Finally, snooping accesses second-level cache tags, which may cause unnecessary contention with processor accesses if the snoops are not often successful in achieving cache-to-cache sharing. Nonetheless, several directory-based systems use snoop-based coherent multiprocessors as their individual nodes (Lenoski et al. 1993; Lovett and Clapp 1996; Clark and Alnes 1996; Weber et al. 1997).

The final approach to improving protocol performance—improving the performance parameters of the communication architecture—is discussed in Section 8.7.

8.4.2 Correctness

As with snoop-based systems, correctness considerations can be divided into three classes. First, the protocol must ensure that the relevant blocks are invalidated/updated and retrieved as needed and that the necessary state transitions occur. We can assume this happens in all cases and not consider it much further. Second, the serialization and ordering relationships defined by coherence and the consistency model must be preserved. Third, the protocol and implementation must be free from deadlock, livelock, and, ideally, starvation. Several aspects of scalable protocols and systems complicate the latter two sets of issues beyond what we have seen for bus-based cache-coherent machines or scalable noncoherent machines. There are two basic problems. First, we now have multiple cached copies of a block but no single agent that can see all relevant transactions and serialize them. Second, with many processors, a large number of requests may be directed toward a single node, accentuating the input buffer problem discussed in Chapter 7. These problems are aggravated by the high latencies in the system, which push us to exploit protocol optimizations of the sort discussed previously; these optimizations allow more transactions to be in progress simultaneously and do not preserve a strict request-response nature, further complicating correctness. This subsection describes the major new issues and types of solutions that are commonly employed in each area of correctness. Some specific solutions used in the case study protocols are discussed in more detail in subsequent sections.

Serialization to a Location for Coherence

Recall the write serialization clause of coherence. Not only must a given processor be able to construct a serial order out of all the operations to a given location—at

least out of all write operations and its own read operations—but all processors must see the writes to a given location as having happened in the same order.

One mechanism we need for serialization is an entity that sees the necessary memory operations to a given location from different processors (the operations that are not contained entirely within a processing node) and determines their serialization. In a bus-based system, operations from different processors are serialized by the order in which their requests appear on the bus. In a distributed system that does not cache shared data, the consistent serializer for a location is the main memory that is the home of a location. For example, the order in which writes become visible to all processors is the order in which they reach the memory, and which write's value a read sees is determined by when that read reaches the memory. In a distributed system with coherent caching, the home memory is again a likely candidate for the entity that determines serialization to a given location, at least in a flat directory scheme, since all relevant operations first come to the home. If the home could satisfy all requests itself, then it could simply process them one by one in FIFO order of arrival and determine serialization. However, with multiple copies, visibility of an operation to the home does not imply visibility to all processors. It is easy to construct scenarios where processors may see operations to a location appear to be serialized in different orders than that in which the requests reached the home, as well as where different processors see operations complete in different orders.

As a simple example, consider an update-based protocol and a network that does not preserve point-to-point order of transactions between the same endpoints. If two write requests for shared data arrive at the home in one order, the updates they generate may arrive at the copies in different orders. As another example, suppose a block is in modified state in a dirty node and two nodes issue read-exclusive requests for it in an invalidation-based protocol. In a strict request-response protocol, the home will provide the requestors with the identity of the dirty node, and they will send requests to it. However, with different requestors, even in a network that preserves point-to-point order there is no guarantee that the requests will reach the dirty node in the same order as they reached the home. Which entity provides the globally consistent serialization in this case, and how is this orchestrated when multiple operations for this block may be simultaneously in flight and potentially needing service from different nodes?

Several types of solutions can be used to ensure serialization to a location. Most of them use additional directory states called *busy states* or *pending states*. A block being in busy state at the directory indicates that a previous request that came to the home for that block is still in progress and has not been completed. When a new request comes to the home and finds the directory state to be busy, serialization may be provided by one of the following mechanisms.

- *Buffer at the home.* The request may be buffered at the home as a pending request until the previous request that is in progress for the block has completed, regardless of whether the previous request was forwarded to a dirty node or whether a strict request-response protocol was used (the home should, of course, process requests for other blocks in the meantime). This method ensures that requests will be serviced everywhere in FIFO order of

their arrival at the home, but it reduces concurrency. It also requires that the home be notified when a write has completed or, more commonly, when the home's involvement with the write is over. Finally, it increases the danger of the input buffer at the home overflowing since this buffer holds pending requests for all blocks for which it is the home. One strategy in this case is to let the input buffer overflow into main memory, thus providing effectively infinite buffering as long as there is enough main memory and avoiding potential deadlock problems. This scheme is used in the MIT Alewife prototype (Agarwal et al. 1995).

- *Buffer at the requestors.* Pending requests may be buffered not at the home but at the requestors themselves, by constructing a distributed linked list of pending requests. This is a natural extension of a cache-based approach, which already has the support for distributed linked lists. It is used in the SCI protocol (Gustavson 1992; IEEE Computer Society 1993). Now the number of pending requests that a node may need to keep track of is small and determined only by the node itself.
- *NACK and retry.* An incoming request may be NACKed by the home (i.e., a negative acknowledgment sent to the requestor) rather than buffered when the directory state is busy. The request will be retried later by the requestor's assist and will be serialized in the order in which it is actually accepted by the directory (attempts that are NACKed do not enter in the serialization order). This is the approach used in the Origin2000 (Laudon and Lenoski 1997).
- *Forward to the dirty node.* If the directory state is busy because a request has been forwarded to a dirty node, subsequent requests for that block are not buffered at the home or NACKed. Rather, they too are forwarded to the dirty node, which determines their serialization. The order of serialization is thus determined by the home node when the block is clean at the home and by the order in which requests reach the dirty node when the block is dirty. If the block in the dirty node leaves the dirty state before a forwarded request reaches it (for example, due to a write back or a previous forwarded request), the request may be NACKed by the dirty node and retried. It will be serialized at the home or a dirty node when the retry is successful. This approach was used in the Stanford DASH protocol (Lenoski et al. 1990; Lenoski et al. 1993).

Unfortunately, with multiple copies in a distributed network, simply identifying a serializing entity is not enough. The problem is that the home or serializing agent may know (or be informed) when its involvement with a request is done, but this does not mean that the request has completed with respect to other nodes. Some transactions for the next request to that block may reach other nodes and perform with respect to them before some remaining transactions for the previous request. We see concrete examples and solutions in our case study protocols in Sections 8.5 and 8.6. Essentially, these show that, in addition to the system providing a global serializing entity for a block, individual nodes (e.g., requestors) should also preserve a local serialization with respect to each block; for example, they should not apply an incoming transaction to a block while they still have a transaction outstanding for that block.

Serialization across Locations for Sequential Consistency

Recall the two most interesting components of preserving the sufficient conditions for satisfying sequential consistency (SC): detecting write completion (needed to preserve program order) and ensuring write atomicity. In a bus-based machine, we saw that the restricted nature of the interconnect allows the requestor to detect write completion early; the write commits and can be acknowledged to the processor as soon as it obtains access to the bus, without waiting for it to actually invalidate or update other caches (Chapter 6). By providing a centralized path through which all transactions pass and ensuring FIFO ordering in the visibility of new data values beyond that path, a bus-based system also makes write atomicity quite natural to ensure.

In a machine that has a distributed network but does not cache shared data, detecting the completion of a write requires an explicit acknowledgment from the memory that holds the location (Chapter 7). In fact, the acknowledgment can be generated early, once we know the write has reached that node and been inserted in a FIFO queue to memory; at this point, the write has committed since it is clear that all subsequent reads that enter the queue will no longer see the old value, and we can use commitment as a substitute for completion to preserve program order. Write atomicity falls out naturally: a write is visible only when it reaches main memory, and at that point it is visible to all processors.

With both multiple copies and a distributed network, it is difficult to assume write completion before the invalidations or updates have actually reached all the nodes. A write cannot be acknowledged to the requestor once it has reached the home and be assumed to have effectively completed. The reason is that a subsequent write Y in program order may be issued by the same requestor after receiving such an acknowledgment for a previous write X, but Y may become visible to another processor before X, thus violating SC. This may happen because the invalidation or update transactions corresponding to Y take a different path through the network or because the network does not provide point-to-point order. Completion, or commitment, can only be assumed once explicit acknowledgments are received from all copies. Of course, a node with a copy can generate the acknowledgment as soon as it receives the invalidation—before it is actually applied to the caches—as long as it guarantees the appropriate ordering within its cache hierarchy (just as commitment is used instead of completion in Chapter 6). To satisfy the sufficient conditions for SC, a processor may wait after issuing a write until all acknowledgments for that write have been received and only then proceed past the write to a subsequent memory operation.

Write atomicity is similarly difficult when there are multiple copies and a distributed interconnect. To see this, Figure 8.14 shows how the semantics assumed by an example code fragment from Chapter 5 (Figure 5.11) that relies on write atomicity can be violated. The constraints of sequential consistency have to be satisfied by orchestrating network transactions appropriately. A common solution for write atomicity in an invalidation-based scheme is for the current owner of a block (the main

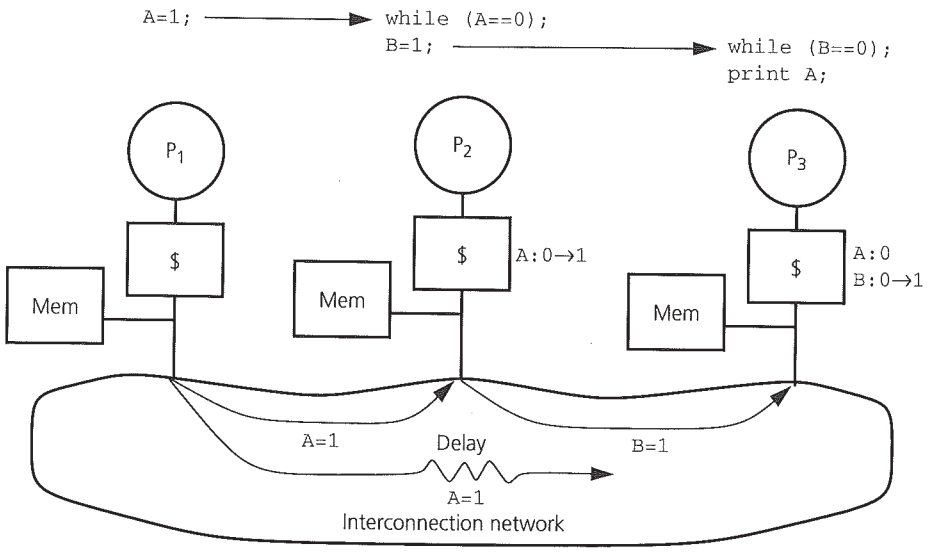


FIGURE 8.14 Violation of write atomicity in a scalable system with caches. The figure shows three processors and the code fragments that they execute. Assume that the network preserves point-to-point order and every cache starts out with copies of A and B initialized to 0. Transactions to look up directories and to satisfy read misses are ignored for simplicity. Under SC, we expect P_3 to print 1 as the value of A. However, P_2 sees the new value of A and jumps out of its while loop to write B even before it knows whether the previous write of A by P_1 has become visible to P_3. This write of B becomes visible to P_3 before the write of A by P_1, because the invalidation or update corresponding to the latter was delayed in a congested part of the network (that the other transactions did not have to go through at all). Thus, P_3 reads the new value of B but the old value of A, yielding a nonintuitive result.

memory module or the processor holding the dirty copy in its cache) to provide the appearance of atomicity by not allowing access to the new value by any process until all invalidation acknowledgments for the write that generated that value have returned. Thus, no processor can see the new value until it is visible to all processors. Maintaining the appearance of atomicity is much more difficult for update-based protocols since the data is sent to the sharers and, hence, is accessible immediately. Ensuring that no sharer reads the value until it is visible to all sharers requires a two-phase interaction. In the first phase, the copies of that memory block are updated in all relevant processors' caches, but those processors are prohibited from accessing the new value. In the second phase, after the first phase is known to have completed through acknowledgments as above, those processors are sent messages that allow them to use the new value. This difficulty and its performance implications help to make update protocols less attractive for scalable directory-based machines than for bus-based machines.

Serialization across Locations for Sequential Consistency

Recall the two most interesting components of preserving the sufficient conditions for satisfying sequential consistency (SC): detecting write completion (needed to preserve program order) and ensuring write atomicity. In a bus-based machine, we saw that the restricted nature of the interconnect allows the requestor to detect write completion early; the write commits and can be acknowledged to the processor as soon as it obtains access to the bus, without waiting for it to actually invalidate or update other caches (Chapter 6). By providing a centralized path through which all transactions pass and ensuring FIFO ordering in the visibility of new data values beyond that path, a bus-based system also makes write atomicity quite natural to ensure.

In a machine that has a distributed network but does not cache shared data, detecting the completion of a write requires an explicit acknowledgment from the memory that holds the location (Chapter 7). In fact, the acknowledgment can be generated early, once we know the write has reached that node and been inserted in a FIFO queue to memory; at this point, the write has committed since it is clear that all subsequent reads that enter the queue will no longer see the old value, and we can use commitment as a substitute for completion to preserve program order. Write atomicity falls out naturally: a write is visible only when it reaches main memory, and at that point it is visible to all processors.

With both multiple copies and a distributed network, it is difficult to assume write completion before the invalidations or updates have actually reached all the nodes. A write cannot be acknowledged to the requestor once it has reached the home and be assumed to have effectively completed. The reason is that a subsequent write Y in program order may be issued by the same requestor after receiving such an acknowledgment for a previous write X, but Y may become visible to another processor before X, thus violating SC. This may happen because the invalidation or update transactions corresponding to Y take a different path through the network or because the network does not provide point-to-point order. Completion, or commitment, can only be assumed once explicit acknowledgments are received from all copies. Of course, a node with a copy can generate the acknowledgment as soon as it receives the invalidation—before it is actually applied to the caches—as long as it guarantees the appropriate ordering within its cache hierarchy (just as commitment is used instead of completion in Chapter 6). To satisfy the sufficient conditions for SC, a processor may wait after issuing a write until all acknowledgments for that write have been received and only then proceed past the write to a subsequent memory operation.

Write atomicity is similarly difficult when there are multiple copies and a distributed interconnect. To see this, Figure 8.14 shows how the semantics assumed by an example code fragment from Chapter 5 (Figure 5.11) that relies on write atomicity can be violated. The constraints of sequential consistency have to be satisfied by orchestrating network transactions appropriately. A common solution for write atomicity in an invalidation-based scheme is for the current owner of a block (the main

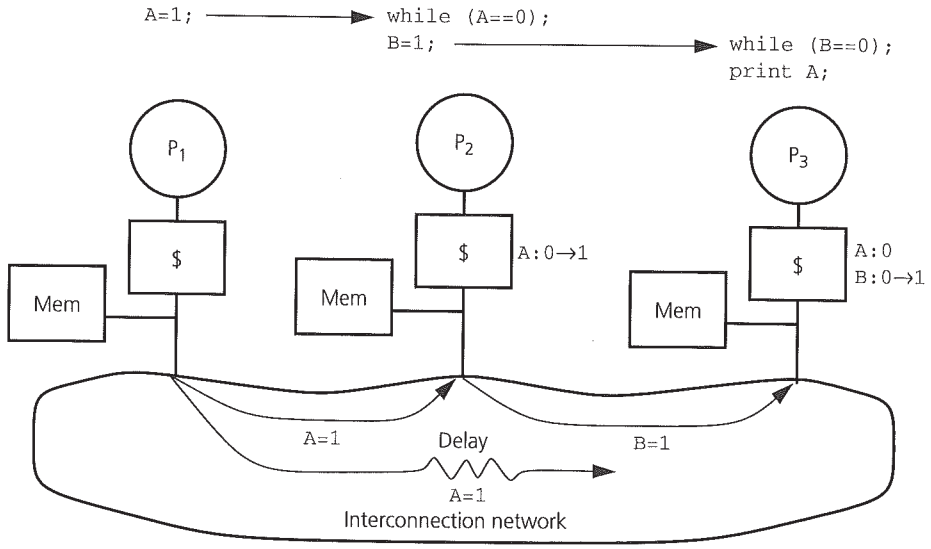


FIGURE 8.14 Violation of write atomicity in a scalable system with caches. The figure shows three processors and the code fragments that they execute. Assume that the network preserves point-to-point order and every cache starts out with copies of A and B initialized to 0. Transactions to look up directories and to satisfy read misses are ignored for simplicity. Under SC, we expect P_3 to print 1 as the value of A. However, P_2 sees the new value of A and jumps out of its while loop to write B even before it knows whether the previous write of A by P_1 has become visible to P_3. This write of B becomes visible to P_3 before the write of A by P_1, because the invalidation or update corresponding to the latter was delayed in a congested part of the network (that the other transactions did not have to go through at all). Thus, P_3 reads the new value of B but the old value of A, yielding a nonintuitive result.

memory module or the processor holding the dirty copy in its cache) to provide the appearance of atomicity by not allowing access to the new value by any process until all invalidation acknowledgments for the write that generated that value have returned. Thus, no processor can see the new value until it is visible to all processors. Maintaining the appearance of atomicity is much more difficult for update-based protocols since the data is sent to the sharers and, hence, is accessible immediately. Ensuring that no sharer reads the value until it is visible to all sharers requires a two-phase interaction. In the first phase, the copies of that memory block are updated in all relevant processors' caches, but those processors are prohibited from accessing the new value. In the second phase, after the first phase is known to have completed through acknowledgments as above, those processors are sent messages that allow them to use the new value. This difficulty and its performance implications help to make update protocols less attractive for scalable directory-based machines than for bus-based machines.

Deadlock

In Chapter 7, we discussed an important source of potential deadlock in request-response protocols such as those of a shared address space: the filling up of a finite input buffer. Three solutions were proposed for buffer deadlock:

1. Provide enough buffer space, either by buffering requests at the requestors using distributed linked lists or by providing enough input buffer space (in hardware or main memory) for the maximum number of possible incoming transactions.
2. Use NACKs.
3. Provide separate request and response networks, whether physically separate or multiplexed with separate buffers, to prevent backups in the potentially poorly behaved request network from blocking the progress of well-behaved response transactions.

Two separate networks would suffice in a protocol that is strictly request-response; that is, in which all transactions can be separated into requests and responses such that a request transaction generates only a response (or nothing) and a response generates no further transactions (and is, in this sense, better behaved since it does not generate further dependences). However, we have seen that in the interest of performance many practical coherence protocols use forwarding and are not always strictly request-response, breaking the deadlock avoidance assumption. In general, we need as many networks (physical or virtual) as the longest chain of different transaction types needed to complete a given operation so that the transaction at the end of a chain (that does not generate further transactions) is always guaranteed to make progress. However, using multiple networks is expensive and many of them will be underutilized. In addition to the approaches that provide enough buffering (as in the HAL S1 and MIT Alewife) or use NACKs throughout, two different approaches deal with deadlock in protocols that are not strict request-response. Both initially pretend that the protocol is strict request-response and provide two real or virtual networks, then rely on detecting situations when deadlock appears possible and resort to a different mechanism to avoid deadlock in these cases. That mechanism may be NACKs or reverting to a strict request-response protocol.

The detection of potential deadlock situations may be done in many ways. In the Stanford DASH machine, a node conservatively assumes that deadlock may be about to happen when both its input request and output request buffers fill up beyond a threshold and the request at the head of the input request buffer is one that may need to generate further requests like interventions or invalidations (i.e., that request is a violator of strict request-response operation and hence capable of causing deadlock). An alternative strategy is to assume the potential for deadlock when the output request buffer is full and has not had a transaction removed from it for T cycles. When potential deadlock is detected, the DASH system takes the first, NACK-based approach to avoiding deadlock: the node takes such requests off from the head of the input queue one by one and sends NACK messages back for them to

the requestors. It does this until the request at the head is no longer one that can generate further requests or until it finds that the output request queue is no longer full. The NACKed requestors will retry their requests later.

A different deadlock avoidance approach is taken by the Origin2000. When potential deadlock is detected, instead of sending a NACK to the requestor, the node sends it a response asking it to send the intervention or invalidation requests directly to the sharers; that is, the system dynamically backs off from a forwarding protocol to a strict request-response protocol, compromising performance temporarily but not allowing deadlock cycles. The advantage of this approach is that NACKing is a statistical rather than robust solution to such congestion-related problems: requests may have to be retried several times in bad situations, leading to increased network traffic and increased latency to the time the operation completes. Dynamic backoff also has advantages related to livelock, as we shall see next.

Livelock

In protocols that avoid deadlock by providing enough buffering of requests, whether centralized or through distributed linked lists, livelock and starvation are taken care of automatically as long as the buffers are FIFO. The other cases do not, in themselves, address livelock and starvation. In these cases, the classic livelock problem due to the race condition of multiple processors trying to write a block at the same time is often taken care of by letting the first request to get to the home go through but NACKing all the others.

NACKs are useful mechanisms for resolving race conditions like the preceding without livelock. However, when used to avoid deadlock in the face of input buffering limitations, as in the DASH solution outlined previously, they have, in fact, the potential to cause livelock. For example, when the node that detects a possible deadlock situation NACKs some requests, it is possible that all those requests are retried at the same time. With extreme pathology, the same situation could repeat itself continually and livelock could result.[2] The alternative solution to deadlock, of switching to a strict request-response protocol in potential deadlock situations, does not cause this livelock problem. It guarantees forward progress and removes the request-request dependence at the home once and for all.

Starvation

The occurrence of starvation is unlikely in well-designed protocols; however, it is not ruled out as a possibility. The fairest solution to starvation is to buffer all requests in FIFO order, which also solves deadlock and livelock. However, this can

2. While the DASH architecture is designed to use NACKs, the actual prototype implementation steps around this problem by using a large enough request input buffer since both the number of nodes and the number of possible outstanding requests per node are small. However, this is not a robust solution for larger, more aggressive machines that cannot provide enough buffer space.

have performance disadvantages, and for protocols that do not do this, avoiding starvation can be difficult to guarantee. Deadlock or livelock solutions that use NACKs and retries are often more susceptible to starvation, which is most likely when many processors repeatedly compete for a resource. Some may keep succeeding while one or more may be very unlucky in their timing and may always get NACKed.

A protocol could decide to do nothing about starvation and rely on the variability of delays in the system not to allow such an indefinitely repeating pathological situation to occur. The DASH machine uses this solution and times out with a bus error if the situation persists beyond a threshold time. Alternatively, a random delay can be inserted between retries to further reduce the small probability of starvation. Finally, requests may be assigned priorities based on the number of times they have already been NACKed, a technique that is used in the Origin2000 protocol.

Having an understanding of the basic directory organizations and high-level protocols as well as the key performance and correctness issues in a general context, we are now ready to dive into actual case studies of memory-based and cache-based protocols. We will see what protocol states and activities look like in actual realizations, how directory protocols interact with and are influenced by the underlying processing nodes, what scalable cache-coherent machines look like, and how actual protocols trade off performance with the complexity of maintaining correctness and of debugging or validating the protocol.

8.5 MEMORY-BASED DIRECTORY PROTOCOLS: THE SGI ORIGIN SYSTEM

Our discussion begins with flat, memory-based directory protocols, using the SGI Origin architecture as a case study. At least for moderate-scale systems, this machine uses essentially a full bit vector directory representation. A similar directory representation but slightly different protocol was also used in the Stanford DASH research prototype (Lenoski et al. 1990), which was the first distributed-memory machine to incorporate directory-based coherence. We follow a similar discussion template for both this and the next case study (the SCI protocol as used in the Sequent NUMA-Q). We begin with the basic coherence protocol, including the directory structure, the directory and cache states, how operations such as reads, writes, and write backs are handled, and the performance enhancements used. Then we will briefly discuss the position taken on the major correctness issues, followed by some prominent protocol extensions for extra functionality. Next, we will examine the rest of the machine as a multiprocessor and how the coherence machinery fits into it. This includes the processing node, the interconnection network, the input/output system, and any interesting interactions between the directory protocol and the underlying node. The case study ends with some important implementation issues (illustrating how it all works and the important data and control pathways), the basic performance characteristics (latency, occupancy, bandwidth) of the protocol, and the resulting application performance for our sample applications.

8.5.1 Cache Coherence Protocol

The Origin system is composed of a number of processing nodes connected by a switch-based interconnection network (see Figure 8.15). Every processing node contains two MIPS R10000 processors, each with first- and second-level caches, a fraction of the total main memory on the machine, an I/O interface, and a single-chip communication assist or coherence controller, called the Hub, that implements the coherence protocol. The Hub is integrated into the memory system. It sees all (second-level) cache misses issued by the processors in that node, whether they are to be satisfied locally or remotely; it receives transactions coming in from the network (in fact, the Hub implements the network interface); and it is capable of retrieving data from the local processor caches.

In terms of the performance issues discussed in Section 8.4.1, at the protocol level, the Origin2000 uses reply forwarding as well as speculative memory operations in parallel with directory lookup at the home. At the machine organization level, the decision in Origin to have two processors per node is driven mostly by cost: several other components on a node (the Hub, the system bus, and so on) are shared between the processors, thus amortizing their cost while hopefully still providing substantial bandwidth per processor. The Origin designers believed that the latency and bandwidth disadvantages of interacting with a snooping bus within a node outweighed its advantages and chose not to maintain snooping coherence between the two processors within a node. Rather, the SysAD (system address and data) bus is simply a shared physical link that is multiplexed between the two processors in a node. This sacrifices the potential advantage of cache-to-cache sharing within the node but eliminates the latency, occupancy, and cache tag contention added by snooping. In particular, with only two processors per node, the likelihood of successful cache-to-cache sharing is small, so the disadvantages may dominate. With a Hub shared between two processors, the combining of requests to the network (not to the directory protocol) could nonetheless have been supported, but it is not, due to the additional implementation cost. When discussing the protocol in this section, let us assume for simplicity that each node contains only one processor, together with its cache hierarchy, a Hub, and main memory. We consider the impact of using two processors per node on the directory structure and protocol later in this section.

Other than reply forwarding, the most interesting aspects of the Origin protocol are its use of busy states and NACKs to resolve race conditions and provide serialization to a location, its deadlock and livelock solution, the way in which it handles race conditions caused by write backs, and its nonreliance on any order preservation among transactions in the network (not even point-to-point order among transactions between the same endpoint nodes). To show how a complete protocol works in the presence of races as well as to illustrate the performance enhancement techniques used in different cases, we will look at how the Origin puts the techniques together to process read and write operations.

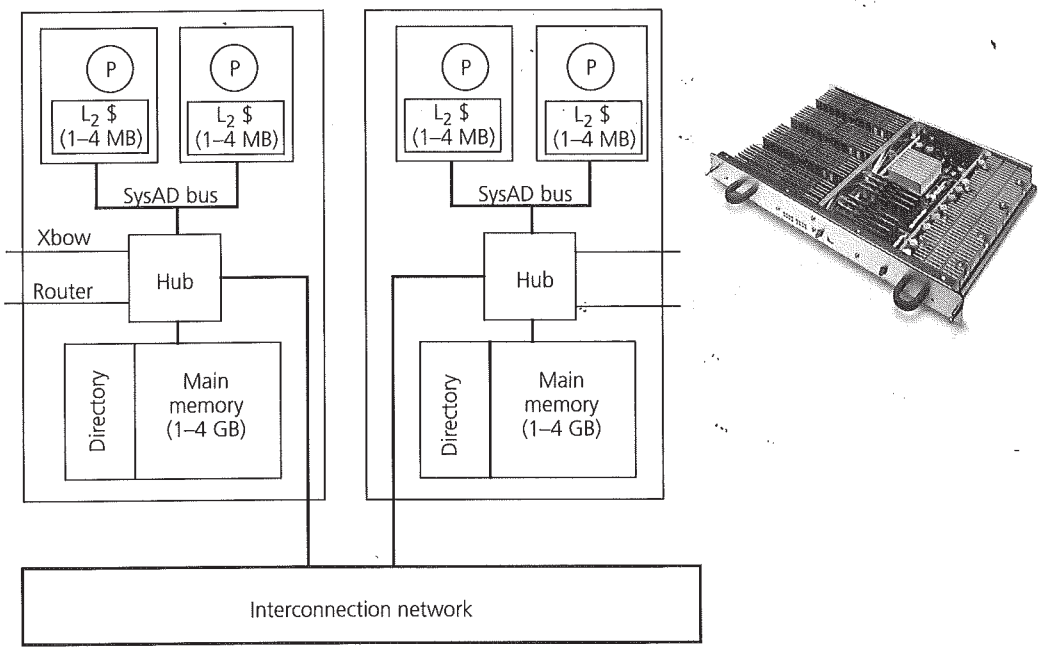


FIGURE 8.15 Block diagram of the Silicon Graphics Origin2000 multiprocessor. Each node contains two processors, a communication assist or controller called the Hub, and main memory with the associated directory. The photograph shows a single node board. *Source:* Photo courtesy of Silicon Graphics, Inc.

Directory Structure and Protocol States

The directory information for a memory block is maintained at the home node for that block. We assume a full bit vector approach for now and examine how the directory organization changes with machine size later.

In the caches, the protocol uses the same MESI states as used in Chapter 5. At the directory, a block may be in one of seven states. Three of these are stable states: *unowned*, or no cached copies in the system; *shared*, that is, zero or more read-only cached copies whose whereabouts are indicated by the presence vector; and *exclusive*, or one read-write cached copy in the system, indicated by the presence vector. An exclusive directory state means the block may be in either dirty or (clean) exclusive state in the cache (i.e., either the M or E states of the MESI protocol). Three other states are *busy* states. As discussed earlier, these imply that the home has received a previous request for that block but was not able to complete that operation itself (e.g., the block may have been dirty in a cache in another node); transactions to complete the request are still in progress in the system, so the directory at the home is not yet ready to handle a new request for that block. The three busy states correspond to three different types of requests that might still be in progress: a read, a read exclusive or upgrade, and an uncached read (a read whose result does

not enter the processor caches and is not kept coherent thereafter). Busy states and NACKs (rather than large amounts of buffering) are used by this protocol to avoid race conditions and provide serialization to a location. The seventh state is a *poison* state, which is used to implement a lazy TLB shootdown method for migrating pages among memories. (Protocol extensions like uncached operations and page migration are discussed in Section 8.5.4.) Given these states, let us see how the coherence protocol handles read, write, and write-back requests from a node.

Handling Read Requests

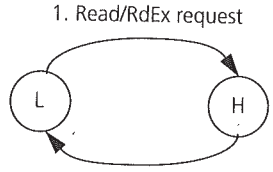
Suppose a processor issues a read that misses in its cache hierarchy. The address of the miss is examined by the local Hub to determine the home node, and a read request transaction is sent to the home node to look up the directory entry. If the home is local, the directory is looked up by the local Hub itself. At the home, the data for the block is accessed speculatively in parallel with looking up the directory entry. The directory entry lookup, which completes a cycle earlier than the speculative data access, may indicate that the memory block is in one of several different states—and different actions are taken in each case.

- *Shared* or *unowned*. This means that main memory at the home has the latest copy of the data (so the speculative access was successful). If the state is shared, the bit corresponding to the requestor is set in the directory presence vector; if it is unowned, the directory state is set to exclusive (achieving the functionality provided by the shared signal in snooping systems). The home then sends the data for the block back to the requestor in a reply transaction. These cases satisfy a strict request-response protocol. Of course, if the home node is the same as the requesting node, then no network transactions or messages are generated and it is a locally satisfied miss.
- *Busy*. This means that the home should not handle the request at this time since a previous request for the block is still in progress. The requestor is sent a negative acknowledge (NACK) message, asking it to try again later. A NACK is categorized as a response, but like an acknowledgment it does not carry data.
- *Exclusive*. This is the most interesting case. If the home is not the owner of the block, the valid data for the block must be obtained from the owner and must find its way to the requestor as well as to the home (since the state will change to shared). The Origin protocol uses reply forwarding; the request is forwarded to the owner, which replies directly to the requestor, sending a revision message to the home. If the home itself is the owner, then the home can simply reply to the requestor, change the directory state to shared, and set the requestor's bit in the presence vector. In fact, in general the directory treats a cache at the home just like any other cache; the only difference is that a "message" between the home directory and a cache at the home does not translate to a network transaction.

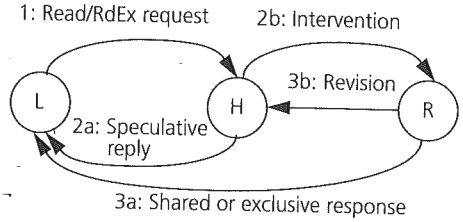
Let us look in a little more detail at what really happens when a read request arrives at the home and finds the state exclusive. (This and several other cases we discuss are illustrated in Figure 8.16.) The main memory block is accessed speculatively in parallel with the directory as usual. When the directory state is discovered to be exclusive, it is set to the busy-exclusive state to deal with subsequent requests, and the request is forwarded to the exclusive node. We cannot set the directory state to shared yet since memory does not yet have an up-to-date copy, and we do not want to leave it as exclusive since then a subsequent request might chase the same exclusive copy of the block as the current request does, requiring that serialization be determined by the current owner node rather than by the home.

Having set the directory entry to a busy state, the presence vector is changed to set the requestor's bit and unset the current owner's. Why this is done at this time becomes clear when we examine write-back requests. Now we see an interesting aspect of the protocol: even though the directory state is exclusive, the home optimistically assumes that the block will be in the (clean) exclusive rather than dirty state in the owner's cache and sends the speculatively accessed memory block at the home as a *speculative reply* (i.e., a reply with data that may or may not be useful) to the requestor. At the same time, the home forwards the intervention request to the owner. The owner checks the state in its cache and performs one of the following actions. If the block is in dirty state, it sends a reply with the data directly to the requestor and a revision message containing the data to the home. At the requestor, the response overwrites the stale speculative reply that was sent by the home. The revision message with data sent to the home is called a *sharing write back* since it writes the data back from the owning cache to main memory and tells it to set the block to shared state. If the block is in exclusive state, the reply to the requestor and the revision message to the home do not contain data since both already have the latest copy (the requestor has it via the speculative reply from the home). The response to the requestor is simply a completion acknowledgment, and the revision message is called a *downgrade* since it asks the home to downgrade the state of the block from (busy) exclusive to shared. In either case, when the home receives the revision message, it changes the state from busy to shared.

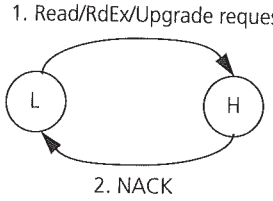
You may have noticed that the use of speculative replies does not have any significant performance advantage in this case since the requestor has to wait to know the real state at the exclusive node anyway before it can use the data. In fact, a simpler alternative to this scheme would be to simply assume that the block is dirty at the owner, not send a speculative reply, and always have the owner send back a reply with the data regardless of whether it has the block in dirty or (clean) exclusive state. Why then does the Origin protocol use speculative replies? There are two reasons, which illustrate how a protocol is influenced by the quirks of existing processors and how different protocol optimizations influence each other. First, the cache controller of the R10000 processor that the Origin uses happens not to return data when it receives an intervention to an exclusive (rather than dirty) cached block since memory is assumed to have a valid copy. Second, speculative replies enable a different optimization in the protocol, which is to allow a processor to simply drop a (clean) exclusive block when it is replaced from the cache, rather than notify main



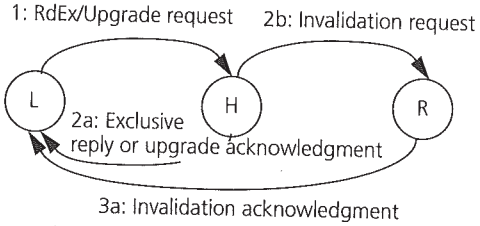
(a) A read or read-exclusive request to a block in unowned state at the directory or a read request to a block in shared state. An exclusive response is sent even in the read case if the block is in unowned state at the directory so that it may be loaded in E rather than S state in the cache.



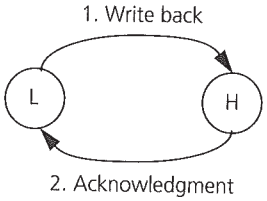
(b) Read or RdEx to a block in the directory in exclusive state. The intervention may be of type shared or exclusive, respectively, with the latter causing invalidation as well. The revision message is a sharing write back or an ownership transfer.



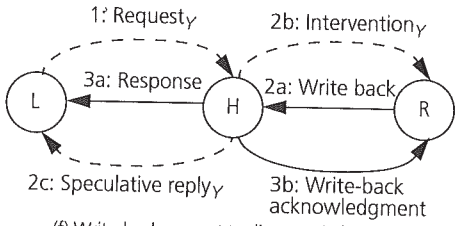
(c) Read/RdEx request to directory in busy state or upgrade request to directory in busy, unowned, or exclusive states.



(d) RdEx or upgrade request to directory in shared state.



(e) Write-back request to directory in exclusive state.



(f) Write-back request to directory in busy state (the Y-subscripted transactions and dashed arcs are those for the other request that made the directory busy).

FIGURE 8.16 Protocol actions in response to requests in the Origin multiprocessor. The case or cases under consideration appear below the diagram, indicating the type of request and the state of the directory entry when the request arrives at the home. The messages or types of transactions are listed next to each arc. Since the same diagram represents different combinations of request type and directory state, different message types are listed on each arc.

memory that it now has the only copy and should reply to subsequent requests since main memory will in any case send a speculative reply when needed.

Handling Write Requests

As we saw in Chapter 5, write misses that invoke the protocol may generate either read-exclusive requests, which request both data and ownership, or upgrade

requests that request only ownership since the requestor's data is valid. In either case, the request goes to the home where the directory state is looked up to determine what actions to take. If the state at the directory is anything but unowned (or busy, which NACKs the request), the copies in other caches must be invalidated. To preserve the ordering model, invalidations must be explicitly acknowledged.

As in the read case, a strict request-response protocol, intervention forwarding, or reply forwarding can be used (see Exercise 8.4). Origin chooses reply forwarding to reduce latency: the home updates the directory state and sends the invalidations directly; it also includes the identity of the requestor in the invalidations so that they are acknowledged directly back to the requestor itself. The actual handling of the read-exclusive and upgrade requests depends on the state of the directory entry when the request arrives; that is, whether it is unowned, shared, exclusive, or busy.

- *Unowned.* If the request is an upgrade, the state at the directory is expected to be shared. The state being unowned means that the block has been replaced from the requestor's cache and the directory notified since it sent the upgrade request (this is possible since the Origin protocol does not assume point-to-point network order). An upgrade is no longer the appropriate request, so it is NACKed. The write operation will be retried, presumably as a read exclusive. If the request is a read exclusive, the directory state is changed to exclusive and the requestor's presence bit is set. The home replies with the data from memory.
- *Shared.* The block must be invalidated in the caches that have copies. The Hub at the home first makes a list of sharers that are to be sent invalidations, using the presence vector. It then sets the directory state to exclusive and sets the presence bit for the requestor. This ensures that the next request for the block will be forwarded to the requestor. If the request was a read exclusive, the home next sends a response to the requestor (called an "exclusive reply with invalidations pending") that also contains the number of sharers from whom to expect invalidation acknowledgments. If the request was an upgrade, the home sends an "upgrade acknowledgment with invalidations pending" to the requestor, which is similar but does not carry the data for the block. In either case, the home next sends invalidation requests to all the sharers, which in turn send acknowledgments to the requestor (not the home). The requestor waits for all acknowledgments to come in before it "closes" or completes the operation. If a new request for the block comes to the home in the meantime, it will see the directory state as exclusive and will be forwarded as an intervention to the current requestor. This current requestor will not handle the intervention immediately but will buffer it until it has received all acknowledgments for its own request and closed that operation. (Further, requests coming to the home in the meantime will find the block in busy-exclusive state, as discussed earlier.)
- *Exclusive.* If the request is an upgrade, then an exclusive directory state means another write has beaten this request to the home. An upgrade is no longer the appropriate request and is NACKed. For a read-exclusive request, the following actions are taken. As with reads, the home sets the directory to a busy

state, sets the presence bit of the requestor, and sends a speculative reply to it. An invalidation request is sent to the owner, containing the identity of the write requestor (if the home is the owner, this is just an invalidation to the local cache and not a network transaction). If the owner has the block in dirty state, it sends a "transfer of ownership" revision message to the home (no data) and a reply with the data to the requestor. This reply overrides the speculative reply that the requestor receives from the home. If the owner has the block in (clean) exclusive state, it relies on the speculative reply from the home and simply sends an acknowledgment to the requestor and a "transfer of ownership" revision message to the home.

- *Busy.* The request is NACKed as in the read case and must try again.

Handling Write-Back Requests and Replacements

When a node replaces a block that is dirty in its cache, it generates a write-back request. This request carries data and is replied to with an acknowledgment by the home. The directory cannot be in unowned or shared state when a write-back request arrives because the write-back requestor has a dirty copy. (A read request cannot change the directory state to shared in between the generation of the write back and its arrival at the home since such a request would have been forwarded to the very node that is requesting the write back and the directory state would have been set to busy.) Let us see what happens when the write-back request reaches the home for the two possible directory states: exclusive and busy.

- *Exclusive.* The directory state transitions from exclusive to unowned (since the only cached copy has been replaced from its cache), and an acknowledgment is returned.
- *Busy.* This indicates an interesting race condition. The directory state can only be busy because an intervention for the block (due to a request from another node *Y*, say) has been forwarded to the very node *X* that is doing the write back. The intervention and write back have crossed each other in the interconnect. Now we are in a funny situation. The other operation from *Y* is already in progress and cannot be undone. We cannot let the write back be dropped, or we would lose the only valid copy of the block. Nor can we NACK the write back and retry it after the operation from *Y* completes, since then *Y*'s cache will have a valid copy while a different dirty copy is being written back to memory from *X*'s cache! This protocol solves the problem by essentially combining the two operations, using the write back as the response to *Y*'s request (see Figure 8.16[f]). The write back that finds the directory state busy changes the state to either shared (if the state was busy-shared, i.e., the request from *Y* was for a read copy) or exclusive (if it was busy-exclusive). The data returned in the write back is then forwarded by the home to the requestor *Y*. This serves as the response to *Y* instead of the response it would have received directly from *X* if there were no write back. When *X* receives an intervention for the block due to *Y*'s request, it simply ignores it (see Exercise 8.13). The directory also

sends a write-back acknowledgment to X. Node Y's operation is complete when it receives the response, and the write back is complete when X receives the write-back acknowledgment. We will see an exception to this treatment in a more complex case when we discuss the serialization of operations. In general, write backs introduce many subtle situations into directory-based coherence protocols.

If the block being replaced from a cache is in shared state, the node may or may not choose to send a replacement hint message back to the home, asking the home to clear its presence bit in the directory. Replacement hints avoid the next useless invalidation to that block and can reduce the occurrence in limited-pointer directory representations, but they incur assist occupancy and do not reduce traffic. In fact, if the block is not written again by another node, then the replacement hint is a waste. The Origin protocol does not use a limited-pointer representation and does not use replacement hints.

In all, the number of transaction types for coherent memory operations in the Origin protocol is 9 requests, 6 invalidations and interventions, and 39 responses. For noncoherent operations such as uncached memory operations, I/O operations, and special synchronization support, the number of transactions is 19 requests and 14 replies (no invalidations or interventions since there is no coherent caching).

8.5.2 Dealing with Correctness Issues

So far, we have seen what happens at different nodes upon read and write misses and how some important race conditions are resolved. Let us now take a different cut through the Origin protocol, examining the specific solutions it adopts for the correctness issues discussed in Section 8.4.2 and the features that the machine provides to deal with errors that may occur.

Serialization to a Location for Coherence

The entity designated to serialize cache misses from different processors is the home. As we have seen, serialization is provided not by buffering requests at the home until previous ones have completed or forwarding them to the owner node even when the directory is in a busy state but by NACKing requests from the home when the state is busy and causing them to be retried. Requests are forwarded only from stable directory states. Serialization is determined by the order in which the home *accepts* the requests—that is, satisfies them itself or forwards them—not the order in which they first arrive at the home.

The general discussion of serialization techniques in Section 8.4.2 suggested that more was needed for serialization to a given location than simply a global serializing entity since the serializing entity does not have full knowledge of when transactions related to a given operation are completed at all the relevant nodes. With a sufficiently in-depth understanding of a protocol, we now examine some concrete examples of this problem (Lenoski 1992) and see how it might be addressed (see Examples 8.1 and 8.2).

EXAMPLE 8.1 Consider the following simple piece of code.

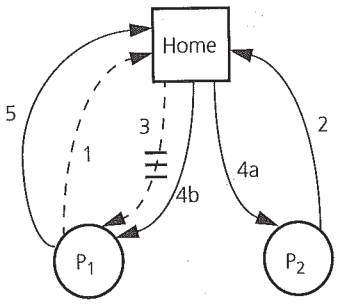
P_1	P_2
rd A (i)	wr A
BARRIER	BARRIER
rd A (ii)	

The write of A may happen either before the first read of A or after it, but it should be serializable with respect to that first read. The second read of A should in any case return the value written by P_2. However, it is quite possible for the effect of the write to get lost if we are not careful. Show how this might happen in a protocol like the Origin's, and discuss possible solutions.

Answer Figure 8.17 shows how the problem can occur, with the text in the figure explaining the transactions, the sequence of events, and the problem. There are two possible solutions. An unattractive one is to have read replies themselves be acknowledged explicitly and let the home go on to process the next request only after it receives this acknowledgment. This further violates the request-response nature of the protocol, causes buffering and potential deadlock problems, and leads to long delays. The more likely solution is to ensure that a node that has a request outstanding for a block, such as P_1, does not allow access by another request, such as the invalidation, to be applied to that block in its cache until its outstanding request completes. P_1 may buffer the incoming invalidation request and apply it only after the read reply is received and completed. Or P_1 can apply the invalidation even before the read reply is received and then consider the reply invalid (a NACK) when it returns and retry the read. Origin uses the former solution whereas the latter is used in DASH. The order of P_1's (first) read with respect to P_2's write is different in the two machines, but both orders are valid. The buffering needed is small and does not cause deadlock problems. ■

EXAMPLE 8.2 In addition to the requestor, the home too may have to disallow new operations from actually being applied to a block (or its directory state) before previous ones have completed as far as it is concerned. Otherwise, directory information may be corrupted. Show an example illustrating this need and discuss solutions.

Answer This example is more subtle and is shown in Figure 8.18. The node issuing the write request detects completion of the write (as far as its involvement is concerned) through acknowledgments before processing another request for the block. The problem is that the home does not wait for its involvement in the write operation—which includes waiting for the revision message and directory update—to complete before it allows another access (here the write back) to be applied to the block. The Origin protocol prevents this from happening by using its busy state: the directory will be in busy-exclusive state when the write back arrives before the revision message. When the directory detects that the write back is coming from the same node whose request put the directory into busy-exclusive state, the write back is NACKed and must be retried. (Recall from the discussion of handling write backs that the write back was treated differently if the request that set the state to busy came from a different node than from the one doing the write back; in that case, the write back was not NACKed but was sent on as the response to the requestor.) ■



1. P_1 sends read request to home node for A.
2. P_2 sends read-exclusive request to home (for the write of A). Home (serializer) won't process it until it is done with read from P_1, which it receives first.
3. In response to (1), home sends reply to P_1 (and sets directory presence bit). Home now thinks read is complete (there are no acknowledgments for a read reply). Unfortunately, the reply does not get to P_1 right away.

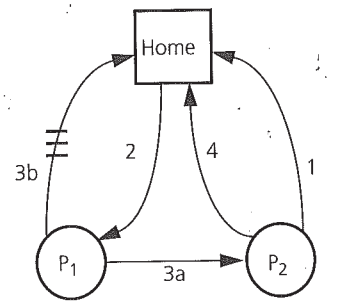
4a. In response to (2), home sends data reply to P_2 corresponding to request 2.

4b. In response to (2), home sends invalidation to P_1; it reaches P_1 before transaction 3 (no point-to-point order is assumed in Origin, and in general the invalidation is a request and 3 is a response, so they may travel on different networks).

5. P_1 receives and applies invalidation, sends acknowledgment to home.

Finally, the read reply (3) reaches P_1 and overwrites the invalidated block. When P_1 reads A after the barrier, it reads this old value rather than seeing an invalid block and fetching the new value. The effect of the write by P_2 is lost as far as P_1 is concerned.

FIGURE 8.17 Example illustrating the need for local serialization of operations at a requestor. The example shows how a write can be lost even though home thinks it is doing things in order. Transactions associated with the first read operation are shown with dotted lines, and those associated with the write operation are shown in solid lines. The three solid bars through a transaction indicate that it is delayed in the network.



Initial condition: block is in dirty state in P_1's cache.

1. P_2 sends read-exclusive request to home.
2. Home forwards request to P_1 (dirty node).
3. P_1 sends data reply to P_2 (3a) and "ownership transfer" revision message to home to change owner to P_2 (3b).
4. P_2, having received its reply, considers write complete. Proceeds, but incurs a replacement of the just dirtied block, causing it to be written back in transaction 4.

This write back is received by the home before the ownership transfer revision message from P_1 (even point-to-point network order wouldn't help), and the block is written into memory. Then when the revision message arrives at the home, the directory is made to point to P_2 as having the dirty copy. But this is untrue, and our protocol is corrupted.

FIGURE 8.18 Example illustrating the need for local serialization of operations at a home node. The example shows how directory information can be corrupted if a home node does not wait for its involvement with a previous request to be over (e.g., a revision message to be received from the owner node) before it allows a new access to the same block.

These examples illustrate the importance of another general requirement that nodes must locally fulfill for proper serialization, beyond the existence of a global serializing entity for a block: any node, not just the serializing entity, should not apply a transaction corresponding to a new memory operation to a block until a previously outstanding memory operation on that block (that the node has begun to handle) is complete as far as that node's involvement is concerned.

Preserving the Memory Consistency Model

The dynamically scheduled R10000 processor allows independent memory operations to issue out of program order, allowing multiple operations to be outstanding at a time and achieving some overlap among them. However, it ensures that operations complete in program order and, in fact, that writes leave the processor environment and become visible to the memory system in program order with respect to other operations, thus preserving sequential consistency (Chapter 11 discusses the necessary processor mechanisms further). The processor does not satisfy the sufficient conditions for sequential consistency spelled out in Chapter 5 in that it does not wait to issue the next operation until the previous one completes, but a system that uses this processor and provides atomicity satisfies the model itself.[3]

Since the processor guarantees visibility and completion in program order, the extended memory hierarchy can perform any reorderings to different locations that it desires without violating this property. The Origin protocol provides write atomicity as discussed earlier: a node does not allow any incoming accesses to a block for which invalidations are outstanding until the acknowledgments for those invalidations have returned (i.e., the write is committed). Nonetheless, one implementation consideration is important in maintaining SC that is due to the Origin protocol's interactions with the processor. Recall from Figure 8.16(d) what happens on a write request (read exclusive or upgrade) to a block that is in shared state at the directory. The requestor receives two types of responses: an *exclusive reply* from the home, discussed earlier, whose role is to indicate that the write has been serialized at memory with respect to other operations for the block and perhaps to return data; and *invalidation acknowledgments,* indicating that the other copies have been invalidated and the write has completed. The microprocessor, however, expects only a single response to its write request, as in a uniprocessor system, so these different responses have to be dealt with by the requesting Hub. To ensure sequential consistency, the Hub must pass the response on to the processor—allowing it to declare completion of the write—only when both the exclusive reply and the invalidation acknowledgments have been received. It must not pass on the response simply when the exclusive reply has been received since that would allow the processor to complete later accesses to other locations even before all invalidations for this one have been

3. This is true for accesses that are under the control of the coherence protocol. The processor also supports memory operations that are not visible to the coherence protocol, called noncoherent memory operations, for which the system does not guarantee any ordering: it is the user's responsibility to insert synchronization to preserve a desired ordering in these cases.

acknowledged, violating sequential consistency. We see in Section 9.1 that such violations are useful when more relaxed memory consistency models than SC are used.

Deadlock, Livelock, and Starvation

The Origin uses finite input buffers and a protocol that is not strict request-response. As discussed in Section 8.4.2, to avoid deadlock, it uses the technique of reverting to a strict request-response protocol when it detects a high-contention situation that may cause deadlock. Since NACKs are not used to alleviate the contention, livelock is avoided in these situations too. The classic livelock problem due to multiple processors trying to write a block at the same time is avoided by using busy states and NACKs (recall that NACKs avoid rather than cause livelock in this case). The first of these requests to get to the home sets the state to busy and makes forward progress while others are NACKed and must retry.

In general, the philosophy of the Origin protocol is twofold: (1) to be "memoryless," that is, every node reacts to incoming events using only current local state and no history of previous events; and (2) not to allow an operation to hold globally shared resources while it is requesting other resources. The latter leads to the choices of NACKing rather than buffering for a busy resource and helps prevent deadlock. These decisions greatly simplify the hardware yet provide high performance in most cases. However, since NACKs are used rather than FIFO ordering, the problem of starvation still exists. This is addressed by associating a priority with a request, which is a function of the number of times the request has been NACKed.[4]

Error Handling

Despite a correct protocol, hardware and software errors can occur at run time. These can corrupt memory or write data to different locations than expected (e.g., if the address on which to perform a write becomes corrupted). The Origin system provides many standard mechanisms to handle hardware errors on components. All caches and memories are protected by error correction codes (ECCs), and all router and I/O links are protected by cyclic redundancy checks (CRCs) and a hardware link-level protocol that automatically detects and retries failures. In addition, the system provides mechanisms to contain failures within the part of the machine in which the program that caused the failure is running. Access protection rights are

4. The priority mechanism works as follows. The directory entry for a block has a "current" priority associated with it. Incoming transactions that will not cause the directory state to become busy are always serviced. Other transactions will potentially be serviced only if their priority is greater than or equal to the current directory priority. If such a transaction is NACKed (e.g., because the directory is in busy state when it arrives), the current priority of the directory is set to be equal to that of the NACKed request. This ensures that the directory will no longer service another request of lower priority until this one is serviced upon retry. To prevent a monotonic increase and "topping out" of the directory entry priority, it is reset to zero whenever a request of priority greater than or equal to it is serviced.

provided on both memory and I/O devices, preventing unauthorized nodes from making modifications. These access rights allow the operating system to be structured into cells or partitions, an organization called a *cellular operating system*. A cell is a number of nodes, configured at boot time. If an application runs within a cell, it may be disallowed from writing memory or I/O outside that cell. If the application fails and corrupts memory or I/O, it can only affect other applications or the system running within that cell and cannot harm code running in other cells. Thus, a cell is the unit of fault containment in the system.

8.5.3 Details of Directory Structure

While we have assumed a full bit vector directory organization so far for simplicity, the actual structure of the Origin directory entry is a little more complex for two reasons: first, to deal with the two processors per node and, second, to allow the directory structure to scale to more than 64 nodes with a 64-bit entry. There are, in fact, three possible formats or interpretations of the directory bits. If a block is in an exclusive state (i.e., modified or exclusive) in a processor cache, then the rest of the directory entry is not a bit vector with one bit turned on but rather contains an explicit pointer to that specific processor (not node). This means that interventions forwarded from the home are targeted to a specific processor. Otherwise, for example, if the directory state is shared, the directory entry is interpreted as a bit vector. Bits in the bit vector correspond to nodes, so even though the two processor caches within a node are not kept coherent by the bus, the unit of visibility to the directory in this format is a node or Hub, not a processor. If an invalidation is sent to a Hub, unlike an intervention, it is broadcast to both processors in the node over the SysAD bus that connects the two processors and the Hub. There are two sizes for presence bit vectors: 16 bit and 64 bit (in the 16-bit case, the directory entry is stored in the same DRAM as the main memory whereas in the 64-bit case the rest of the bits are in an extended directory memory module that is looked up in parallel). The 16-bit vector therefore supports up to 32 processors, and the 64-bit vector supports up to 128 processors.

For larger systems, the interpretation of the bits changes to the third format. In a p-node system, each bit now corresponds to a fixed set of $p/64$ nodes. The bit is set when any one (or more) of the nodes in the corresponding set has a copy of the block. If a bit is set when a write happens, then invalidations are sent to all the $p/64$ nodes represented by that bit (and are then broadcast to both processors in each of those nodes). For example, with the maximum supported size of 1,024 processors (512 nodes), each bit corresponds to 8 nodes. This is called a *coarse vector* representation, and we see it again when we discuss overflow strategies for directory representations as an advanced topic in Section 8.10. In fact, the system dynamically chooses between the bit vector and coarse vector representation on a large machine: if all the nodes sharing the block are within the same 64-node octant of the machine, a bit vector representation is used; otherwise, a coarse vector is used.

8.5.4 Protocol Extensions

In addition to the protocol optimizations discussed earlier, the Origin protocol provides some extensions to support special operations and activities that interact with the protocol. These include input/output and DMA operations, page migration, and synchronization.

Support for Input/Output and DMA Operations

To support memory reads by a DMA device, the protocol provides "uncached read-shared" requests. Such a request returns to the DMA device a snapshot of a coherent copy of the data, but that copy is then no longer kept coherent by the protocol. The request is used primarily by the I/O system and the block transfer engine provided in the Hub and as such is intended for use by the operating system. For writes to memory from a DMA device, the protocol provides "write invalidate" requests. A write invalidate simply blasts the new value of a word into memory, overwriting the previous value. It also invalidates all existing cached copies of the block in the system, thus returning the directory entry to unowned state. From a protocol perspective, it behaves much like a read-exclusive request, except that it modifies the block in memory and leaves the directory in unowned state.

Support for Automatic Page Migration

As we discussed in Chapter 3, on a machine with physically distributed memory it is often important to allocate data appropriately across physical memories so that most capacity, conflict, and cold misses are satisfied locally. On CC-NUMA machines like the Origin, data is allocated in memory at the granularity of a page (16 KB, in this case). Despite the very aggressive communication architecture in the Origin, the latency of an access satisfied by remote memory is at least 2–3 times that of a local access even without contention. The appropriate distribution of pages among memories might change dynamically at run time, either because a parallel program's access patterns change or because the operating system decides to migrate an application process from one processor to another for better resource management across multiprogrammed applications. It is therefore useful for the system to detect the need for moving pages at run time and migrate them automatically to where they are needed.

For every page in main memory, Origin provides an array of miss counters, one per node, to help determine when most of the misses to a page are coming from a nonlocal processor so that the page should be migrated. The miss counters are stored in directory memory at the home. When a request comes in for a page, the miss counter for that node is incremented and compared with the miss counter for the home node. If it exceeds the latter by more than a threshold, then the page can be migrated to that remote node. (Sixty-four counters are provided per page, and in a system with more than 64 nodes, 8 nodes share a counter.) Page migration is typi-