Coherence requests are a significant but not overwhelming component in the scientific processing workload. We can expect, however, that coherence requests will be more important in parallel programs that are less optimized.

The question of how these cache miss rates affect CPU performance depends on the rest of the memory system, including the latency and bandwidth of the interconnect and memory.

## 6.5 Distributed Shared-Memory Architectures

A scalable multiprocessor supporting shared memory could choose to exclude or include cache coherence. The simplest scheme for the hardware is to exclude cache coherence, focusing instead on a scalable memory system. Several companies have built this style of multiprocessor; the Cray T3D/E is the best known example. In such multiprocessors, memory is distributed among the nodes and all nodes are interconnected by a network. Access can be either local or remote—a controller inside each node decides, on the basis of the address, whether the data reside in the local memory or in a remote memory. In the latter case a message is sent to the controller in the remote memory to access the data.

These systems have caches, but to prevent coherence problems, shared data are marked as uncacheable and only private data are kept in the caches. Of course, software can still explicitly cache the value of shared data by copying the data from the shared portion of the address space to the local private portion of the address space that is cached. Coherence is then controlled by software. The advantage of such a mechanism is that little hardware support is required, although support for features such as block copy may be useful, since remote accesses fetch only single words (or double words) rather than cache blocks.

There are several disadvantages to this approach. First, compiler mechanisms for transparent software cache coherence are very limited. The techniques that currently exist apply primarily to programs with well-structured loop-level parallelism or a very strict form of object-oriented programming, and these techniques have significant overhead arising from explicitly copying data. For irregular problems or problems involving dynamic data structures and pointers (including operating systems, for example), compiler-based software cache coherence is currently impractical. The basic difficulty is that software-based coherence algorithms must be conservative: Every block that *might* be shared must be treated as if it *is* shared. Being conservative results in excess coherence overhead because the compiler cannot predict the actual sharing accurately enough. Due to the complexity of the possible interactions, asking programmers to deal with coherence is unworkable.

Second, without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word. The benefits of spatial locality in shared data cannot be leveraged when single words are fetched from a remote memory for each reference. Support for a DMA mechanism among memories can help, but such mech-

anisms are often either costly to use (since they may require OS intervention) or expensive to implement (since special-purpose hardware support and a buffer are needed). For message-passing programs, however, such mechanisms can be extremely useful, since programmers can overcome the usage penalties by using large messages.

Third, mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; we will examine this advantage in more detail later.

These disadvantages are magnified by the large latency of access to remote memory versus a local cache. For example, on the Cray T3E a local cache access has a latency of 2 cycles and is pipelined. A remote memory access takes up to 400 processor clock cycles for a remote memory using the 450 MHz Alpha processor in the T3E-900.

For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors. For larger-scale architectures, there are new challenges to extending the cache-coherent shared-memory model. Although the bus can certainly be replaced with a more scalable interconnection network (e.g., the Sun Enterprise servers use up to four buses), and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed.

A snooping protocol requires communication with all caches on every cache miss, including writes of potentially shared data. The absence of any centralized data structure that tracks the state of the caches is both the fundamental advantage of a snooping-based scheme, since it allows it to be inexpensive, as well as its Achilles' heel when it comes to scalability. For example, with only 16 processors, a block size of 64 bytes, and a 512 KB data cache, the total bus bandwidth demand (ignoring stall cycles) for the four programs in the scientific/technical workload ranges from about 1 GB/sec (for Barnes) to about 42 GB/sec (for FFT), assuming a processor that sustains a data reference every 1 ns, which is what a 2001 superscalar processor with nonblocking caches might generate. In comparison, the Sun Enterprise system with the Starfire interconnect, the highest-bandwidth SMP system in 2001, can support about 12 GB/sec of random accesses for the $16 \times 16$ crossbar and has a maximum bandwidth of 21.3 GB/sec at the memory system. Although the cache size used in these simulations is moderate (but large enough to eliminate much of the uniprocessor miss traffic), so is the problem size.

Alternatively, we could build scalable shared-memory architectures that include cache coherency. The key is to find an alternative coherence protocol to the snooping protocol. One alternative protocol is a *directory protocol*. A directory keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on. (Section 6.11 on page 622 describes a hybrid approach that uses directories to extend a snooping protocol.)

Existing directory implementations associate an entry in the directory with each memory block. In typical protocols, the amount of information is proportional to the product of the number of memory blocks and the number of processors. This

overhead is not a problem for multiprocessors with less than about 200 processors because the directory overhead will be tolerable. For larger multiprocessors, we need methods to allow the directory structure to be efficiently scaled. The methods that have been proposed either try to keep information for fewer blocks (e.g., only those in caches rather than all memory blocks) or try to keep fewer bits per entry.

To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property is what allows the coherence protocol to avoid broadcast. Figure 6.27 shows how our distributed-memory multiprocessor looks with the directories added to each node.

## Directory-Based Cache Coherence Protocols: The Basics

Just as with a snooping protocol, there are two primary operations that a directory protocol must implement: handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.) To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

- *Shared*—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches).

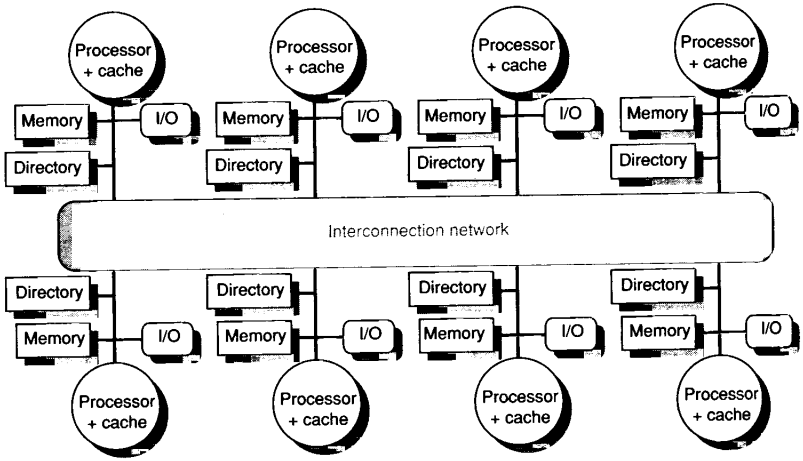


**Figure 6.27 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor.** Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over a common bus, as shown, or it may have a separate port to memory, or it may be part of a central node controller through which all intranode and internode communications pass.

- *Uncached*—No processor has a copy of the cache block.
- *Exclusive*—Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.

In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. We make the same simplifying assumptions that we made in the case of the snooping cache: Attempts to write data that are not exclusive in the writer's cache always generate write misses, and the processors block until an access completes. Since the interconnect is no longer a bus and since we want to avoid broadcast, there are two additional complications. First, we cannot use the interconnect as a single point of arbitration, a function the bus performed in the snooping case. Second, because the interconnect is message oriented (unlike the bus, which is transaction oriented), many messages must have explicit responses.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories. Figure 6.28 shows the type of messages sent among nodes. The *local node* is the node where a request originates. The *home node* is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a *remote node*.

A remote node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

In this section, we assume a simple model of memory consistency. To minimize the type of messages and the complexity of the protocol, we make an assumption that messages will be received and acted upon in the same order they are sent. This assumption may not be true in practice and can result in additional complications, some of which we address in Section 6.8 when we discuss memory consistency models. In this section, we use this assumption to ensure that invalidates sent by a processor are honored immediately.

| Message type | Source | Destination | Message contents | Function of this message |
|---|---|---|---|---|
| Read miss | local cache | home directory | P, A | Processor P has a read miss at address A; request data and make P a read sharer. |
| Write miss | local cache | home directory | P, A | Processor P has a write miss at address A; request data and make P the exclusive owner. |
| Invalidate | home directory | remote cache | A | Invalidate a shared copy of data at address A. |
| Fetch | home directory | remote cache | A | Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared. |
| Fetch/invalidate | home directory | remote cache | A | Fetch the block at address A and send it to its home directory; invalidate the block in the cache. |
| Data value reply | home directory | local cache | D | Return a data value from the home memory. |
| Data write back | remote cache | home directory | A, D | Write back a data value for address A. |

**Figure 6.28 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting processor number, A = requested address, and D = data contents), and the function of the message.** The first two messages are miss requests sent by the local cache to the home. The third through fifth messages are messages sent to a remote cache by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol, since any dirty block must be exclusive and any shared block is always available in the home memory.

## An Example Directory Protocol

The basic states of a cache block in a directory-based protocol are exactly like those in a snooping protocol, and the states in the directory are also analogous to those we showed earlier. Thus we can start with simple state diagrams that show the state transitions for an individual cache block and then examine the state diagram for the directory entry corresponding to each block in memory. As in the snooping case, these state transition diagrams do not represent all the details of a coherence protocol; however, the actual controller is highly dependent on a number of details of the multiprocessor (message delivery properties, buffering structures, and so on). In this section we present the basic protocol state diagrams. The knotty issues involved in implementing these state transition diagrams are examined in Appendix I, along with similar problems that arise for snooping caches.

Figure 6.29 shows the protocol actions to which an individual cache responds. We use the same notation as in the last section, with requests coming from outside the node in gray and actions in bold. The state transitions for an individual cache are caused by read misses, write misses, invalidates, and data fetch requests; these operations are all shown in Figure 6.29. An individual cache also generates read and write miss messages that are sent to the home directory. Read and write misses require data value replies, and these events wait for replies before changing state.

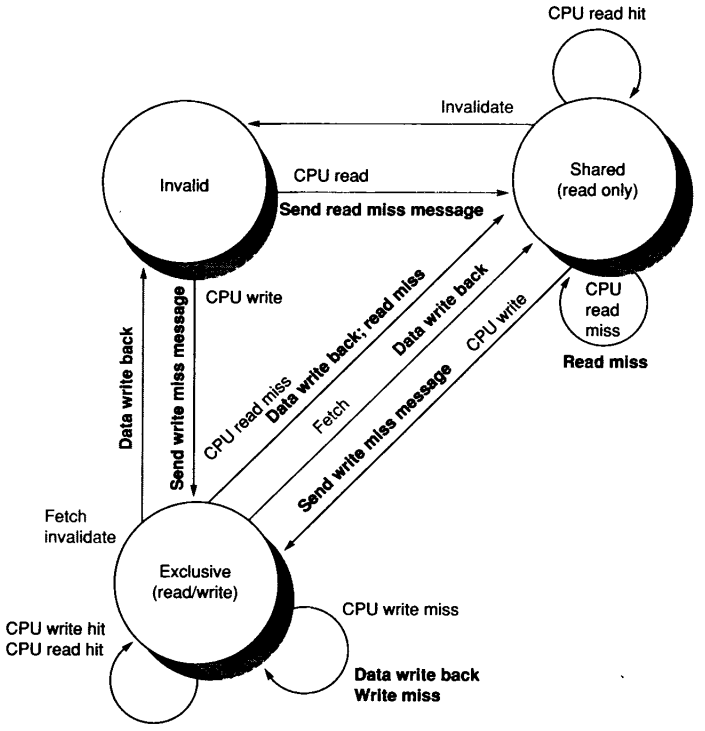


**Figure 6.29 State transition diagram for an individual cache block in a directory-based system.** Requests by the local processor are shown in black and those from the home directory are shown in gray. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

The operation of the state transition diagram for a cache block in Figure 6.29 is essentially the same as it is for the snooping case: The states are identical, and the stimulus is almost identical. The write miss operation, which was broadcast on the bus in the snooping scheme, is replaced by the data fetch and invalidate operations that are selectively sent by the directory controller. Like the snooping protocol, any cache block must be in the exclusive state when it is written, and any shared block must be up to date in memory.

In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of actions: updates of the directory state and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a

block; unlike in a snoopy scheme, however, the directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block. The memory block may be uncached by any node, cached in multiple nodes and readable (shared), or cached exclusively and writable in exactly one node. In addition to the state of each block, the directory must track the set of processors that have a copy of a block; we use a set called *Sharers* to perform this function. In multiprocessors with less than 64 nodes (which may represent two to four times as many processors), this set is typically kept as a bit vector. In larger multiprocessors, other techniques, which we discuss in Exercise 6.16, are needed. Directory requests need to update the set Sharers and also read the set to perform invalidations.

Figure 6.30 shows the actions taken at the directory in response to messages received. The directory receives three different requests: read miss, write miss, and data write back. The messages sent in response by the directory are shown in bold, while the updating of the set Sharers is shown in bold italics. Because all the stimulus messages are external, all actions are shown in gray. Our simplified

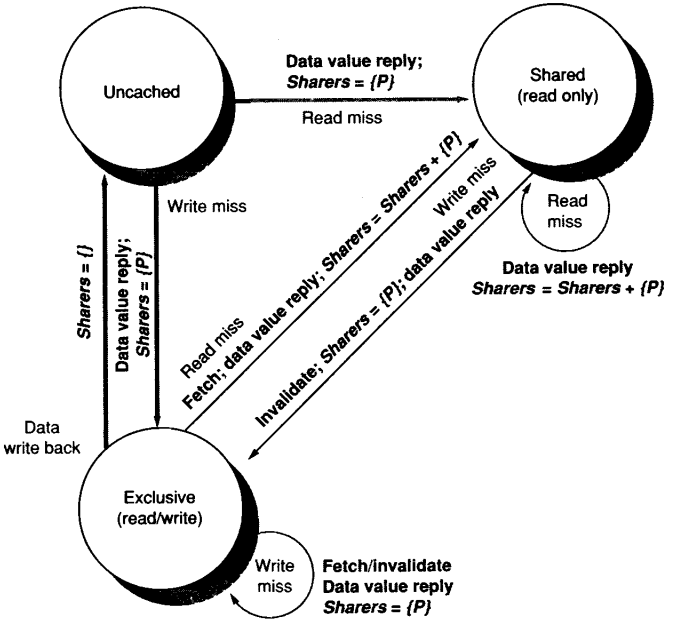


**Figure 6.30 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache.** All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request. Bold italics indicate an action that updates the sharing set, Sharers, as opposed to sending a message.

protocol assumes that some actions are atomic, such as requesting a value and sending it to another node; a realistic implementation cannot use this assumption.

To understand these directory operations, let's examine the requests received and actions taken state by state. When a block is in the uncached state, the copy in memory is the current value, so the only possible requests for that block are

- *Read miss*—The requesting processor is sent the requested data from memory and the requestor is made the only sharing node. The state of the block is made shared.
- *Write miss*—The requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

- *Read miss*—The requesting processor is sent the requested data from memory and the requesting processor is added to the sharing set.
- *Write miss*—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state, the current value of the block is held in the cache of the processor identified by the set Sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
- *Data write back*—The owner processor is replacing the block and therefore must write it back. This write back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

This state transition diagram in Figure 6.30 is a simplification, just as it was in the snooping cache case. In the directory case it is a larger simplification, since our assumption that bus transactions are atomic no longer applies. Appendix I explores these issues in depth.

In addition, the directory protocols used in real multiprocessors contain additional optimizations. In particular, in this protocol when a read or write miss occurs for a block that is exclusive, the block is first sent to the directory at the home node. From there it is stored into the home memory and also sent to the original requesting node. Many of the protocols in use in commercial multiprocessors forward the data from the owner node to the requesting node directly (as well as performing the write back to the home). Such optimizations often add complexity by increasing the possibility of deadlock and by increasing the types of messages that must be handled.

## 6.6 Performance of Distributed Shared-Memory Multiprocessors

The performance of a directory-based multiprocessor depends on many of the same factors that influence the performance of bus-based multiprocessors (e.g., cache size, processor count, and block size), as well as the distribution of misses to various locations in the memory hierarchy. The location of a requested data item depends on both the initial allocation and the sharing patterns. We start by examining the basic cache performance of our scientific/technical workload and then look at the effect of different types of misses.

Because the multiprocessor is larger and has longer latencies than our snooping-based multiprocessor, we begin with a slightly larger cache (128 KB) and a larger block size of 64 bytes.

In distributed-memory architectures, the distribution of memory requests between local and remote is key to performance because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section we separate the cache misses into local and remote requests. In looking at the figures, keep in mind that, for these applications, most of the remote misses that arise are coherence misses, although some capacity misses can also be remote, and in some applications with poor data distribution such misses can be significant (see the pitfall on page 643).

As Figure 6.31 shows, the miss rates with these cache sizes are not affected much by changes in processor count, with the exception of Ocean, where the miss rate rises at 64 processors. This rise results from two factors: an increase in mapping conflicts in the cache that occur when the grid becomes small, which leads to a rise in local misses, and an increase in the number of the coherence misses, which are all remote.

Figure 6.32 shows how the miss rates change as the cache size is increased, assuming a 64-processor execution and 64-byte blocks. These miss rates decrease at rates that we might expect, although the dampening effect caused by little or no reduction in coherence misses leads to a slower decrease in the remote misses than in the local misses. By the time we reach the largest cache size shown, 512 KB, the remote miss rate is equal to or greater than the local miss rate. Larger caches would amplify this trend.