Chapter 5 - Thread-Level Parallelism

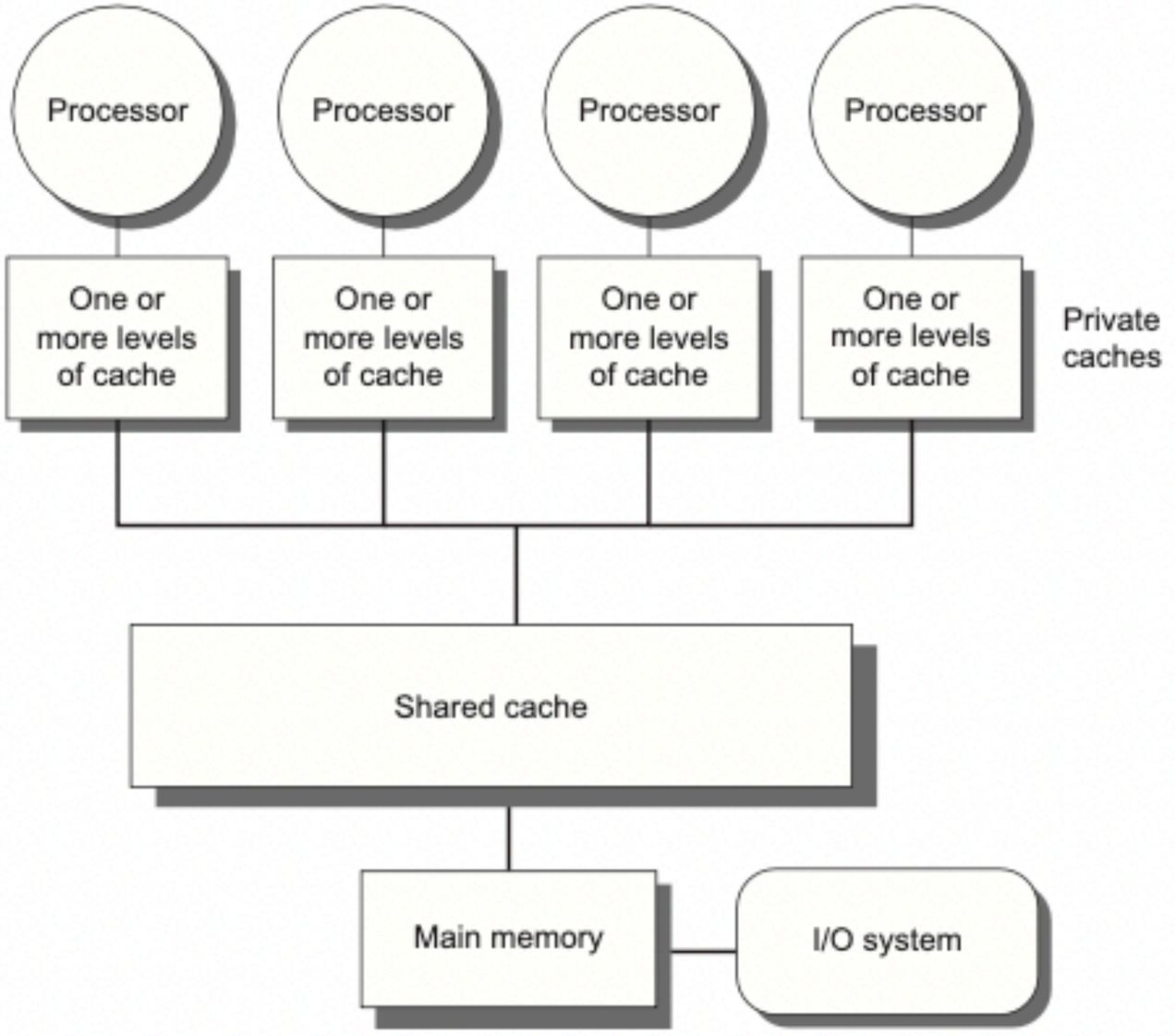

Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical mem-

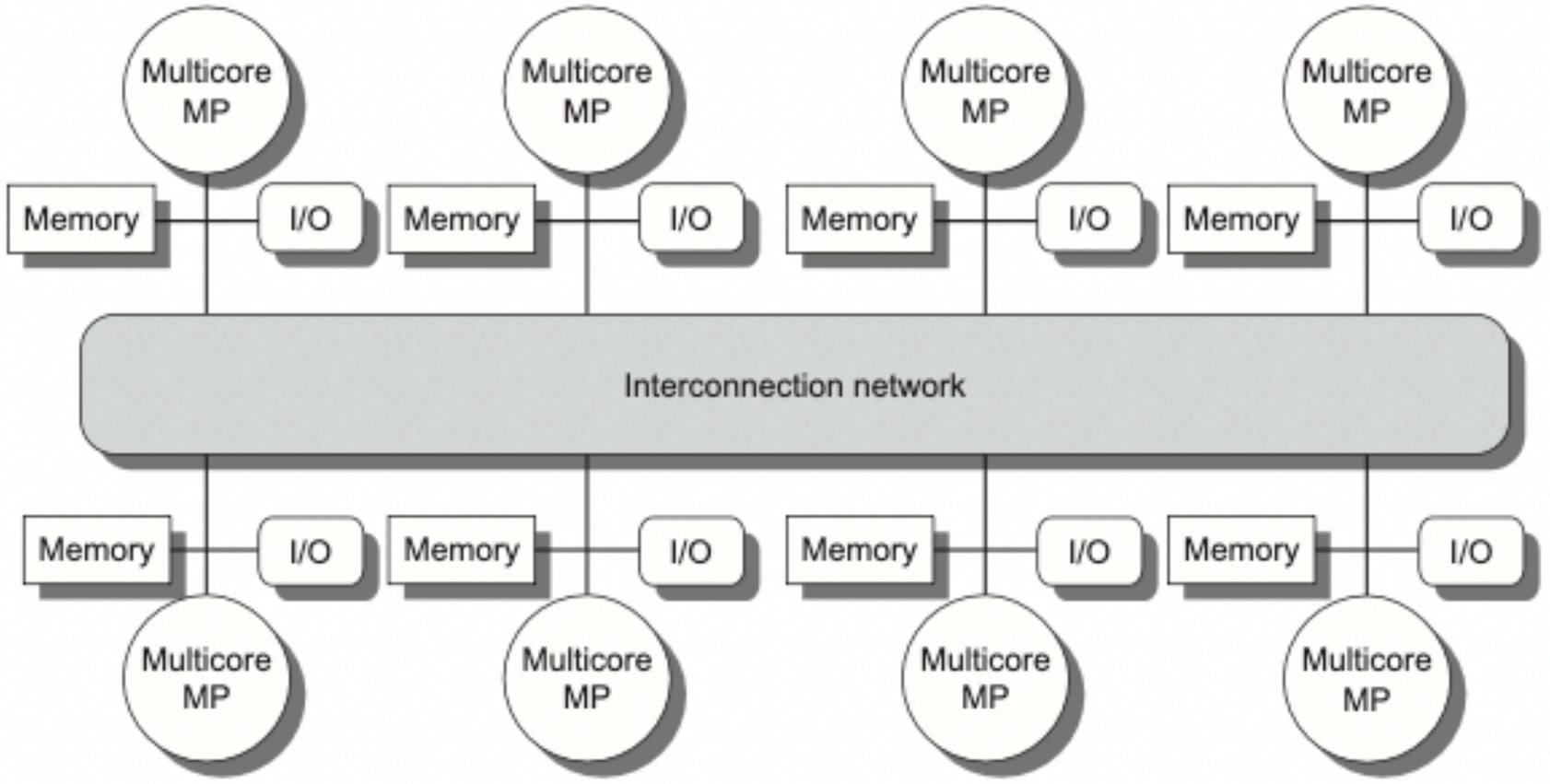

Figure 5.2 The basic architecture of a distributed-memory multiprocessor in 2017 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes. Each processor core shares the entire memory, although the access time to the local memory attached to the core's chip will be much faster than the access time to remote memories.

Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Recall from Chapter 1 that Amdahl's Law is

$$\text{Speedup} = \frac{1}{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, whereas the fraction of enhanced mode is the time spent in parallel mode. Substituting into the previous equation:

$$80 = \frac{1}{\dfrac{\text{Fraction}_{\text{parallel}}}{100} + \left(1 - \text{Fraction}_{\text{parallel}}\right)}$$

Simplifying this equation yields:

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times \left(1 - \text{Fraction}_{\text{parallel}}\right) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of the original computation can be sequential! Of course, to achieve linear speedup (speedup of n with n processors), the entire program must usually be parallel with no serial portions. In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode. Amdahl's Law can be used to analyze applications with varying amounts of speedup, as the next example shows.

Example Suppose we have an application running on a 100-processor multiprocessor, and assume that application can use 1, 50, or 100 processors. If we assume that 95% of the time we can use all 100 processors, how much of the remaining 5% of the execution time must employ 50 processors if we want a speedup of 80?

Answer We use Amdahl's Law with more terms:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{100}}{\text{Speedup}_{100}} + \frac{\text{Fraction}_{50}}{\text{Speedup}_{50}} + (1 - \text{Fraction}_{100} - \text{Fraction}_{50})}$$

Substituting in:

$$80 = \frac{1}{\frac{0.95}{100} + \frac{\text{Fraction}_{50}}{50} + (1 - 0.95 - \text{Fraction}_{80})}$$

Simplifying:

$$0.76 + 1.6 \times \text{Fraction}_{50} + 4.0 - 80 \times \text{Fraction}_{50} = 1$$
$$4.76 - 78.4 \times \text{Fraction}_{50} = 1$$
$$\text{Fraction}_{50} = 0.048$$

If 95% of an application can use 100 processors perfectly, to get a speedup of 80, 4.8% of the remaining time must be spent using 50 processors and only 0.2% can be serial!

Example Suppose we have an application running on a 32-processor multiprocessor that has a 100 ns delay to handle a reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is obviously optimistic. Processors are stalled on a remote request, and the processor clock rate is 4 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

Answer It is simpler to first calculate the clock cycles per instruction. The effective CPI for the multiprocessor with 0.2% remote references is

$$\begin{aligned}\text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost}\\ &= 0.5 + 0.2\% \times \text{Remote request cost}\end{aligned}$$

The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{100\,\text{ns}}{0.25\,\text{ns}} = 400\,\text{cycles}$$

Therefore we can compute the CPI:

$$\begin{aligned}\text{CPI} &= 0.5 + 0.20\% \times 400\\ &= 1.3\end{aligned}$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster. In practice, the performance analysis is much more complex because some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be worse because contention caused by many references trying to use the global interconnect can lead to increased delays, or the access time might be better if memory were distributed and the access was to the local memory.

This problem could have also been analyzed using Amdahl's Law, an exercise we leave to the reader.

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Figure 5.3 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X it will receive 1!

A memory system is coherent if

1. A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

Directory-Based vs Snooping Protocols

- *Directory based*—The sharing status of a particular block of physical memory is kept in one location, called the *directory*. There are two very different types of directory-based cache coherence. In an SMP, we can use one centralized directory, associated with the memory or some other single serialization point, such as the outermost cache in a multicore. In a DSM, it makes no sense to have a single directory because that would create a single point of contention and make it difficult to scale to many multicore chips given the memory demands of multicores with eight or more cores. Distributed directories are more complex than a single directory, and such designs are the subject of Section 5.4.
- *Snooping*—Rather than keeping the state of sharing in a single directory, every cache that has a copy of the data from a block of physical memory could track the sharing status of the block. In an SMP, the caches are typically all accessible via some broadcast medium (e.g., a bus connects the per-core caches to the shared cache or memory), and all cache controllers monitor or *snoop* on the medium to determine whether they have a copy of a block that is requested on a bus or switch access. Snooping can also be used as the coherence protocol for a multichip multiprocessor, and some designs support a snooping protocol on top of a directory protocol within each multicore.

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Figure 5.4 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The pro-

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Figure 5.5 The cache coherence mechanism receives requests from both the core's processor and the shared bus and responds to these based on the type of request, whether it hits or misses in the local cache, and the state of the local cache block specified in the request. The fourth column describes the type of cache action as normal hit or

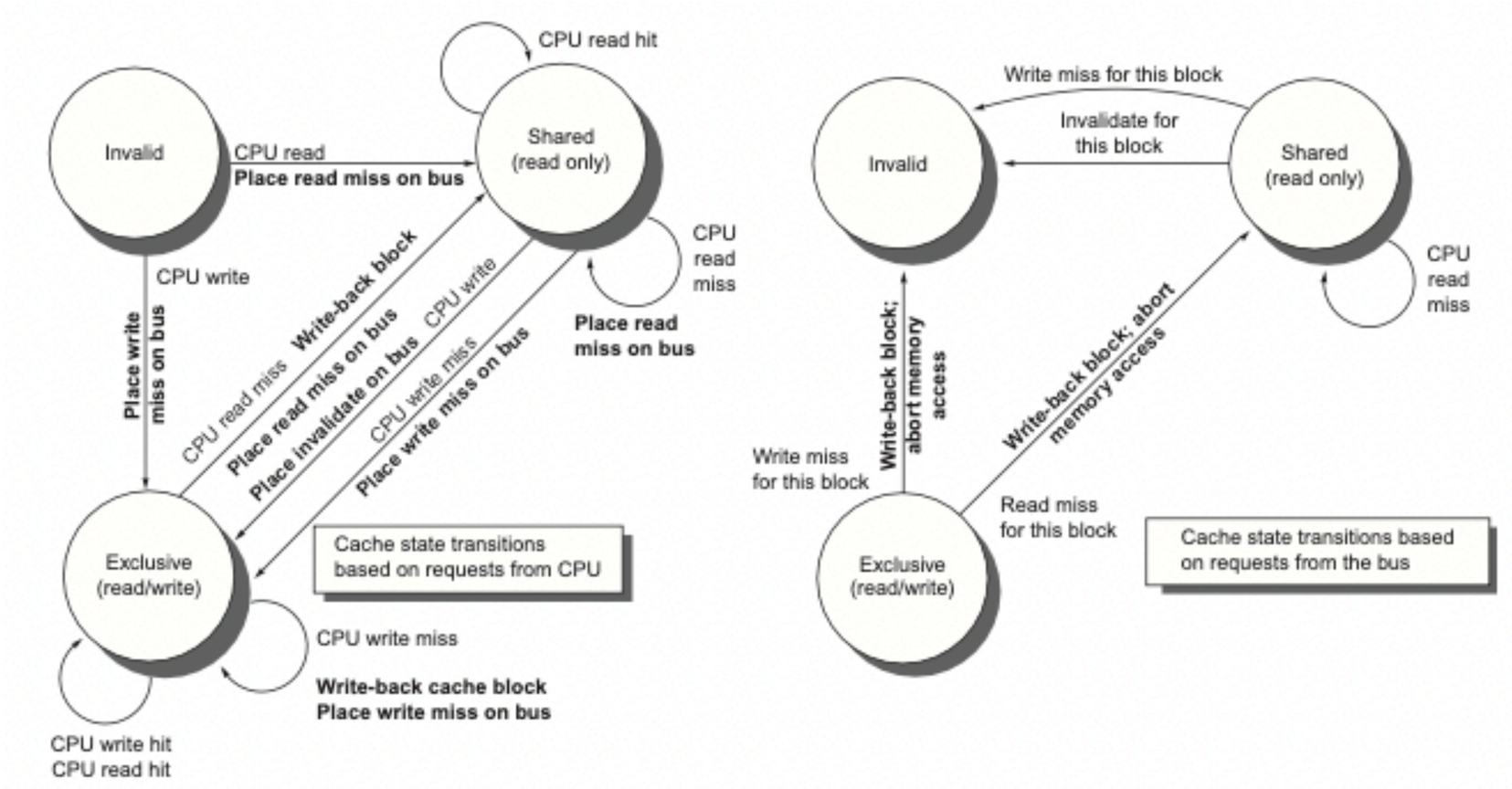

Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the

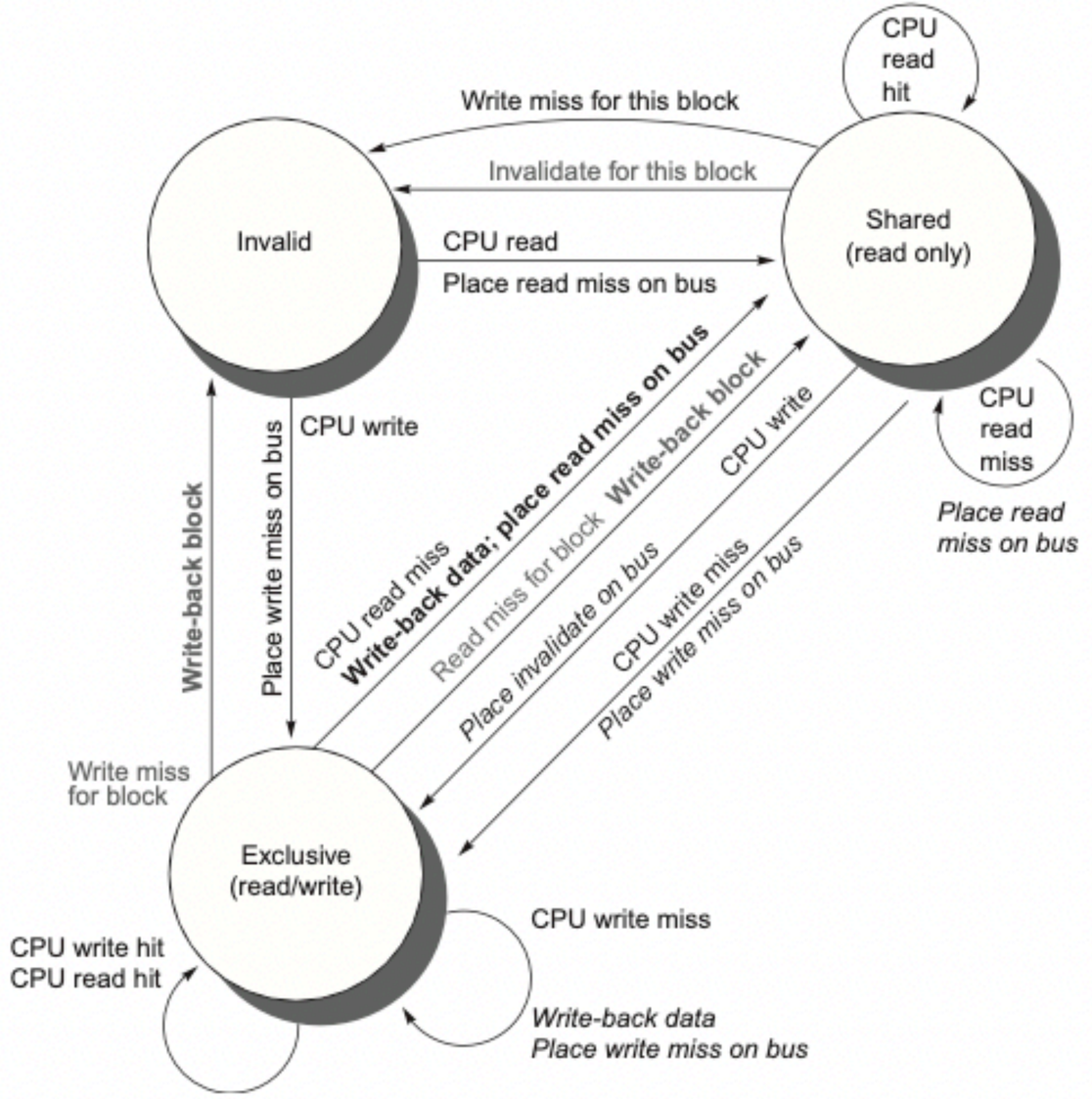

Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in *black* and by the bus activities shown in *gray*. As in Figure 5.6, the activities on a transition are shown in *bold*.

Other Protocol Variations

- MESI (Modified, Exclusive, Shared, Invalid)
  - Exclusive - Block is in only one cache, and is clean
- MOESI (Modified, Owned, Exclusive, Shared, Invalid)
  - Owned - One cache “owns” block, is out-of-date in main memory
- MESIF (Modified, Exclusive, Shared, Invalid, Forward)
  - Forward - Cache in ‘F’ state will be only responder for cache request

Example Consider an 8-processor multicore where each processor has its own L1 and L2 caches, and snooping is performed on a shared bus among the L2 caches. Assume the average L2 request, whether for a coherence miss or other miss, is 15 cycles. Assume a clock rate of 3.0 GHz, a CPI of 0.7, and a load/store frequency of 40%. If our goal is that no more than 50% of the L2 bandwidth is consumed by coherence traffic, what is the maximum coherence miss rate per processor?

Answer Start with an equation for the number of cache cycles that can be used (where CMR is the coherence miss rate):

$$\text{Cache cycles available} = \frac{\text{Clock rate}}{\text{Cycles per request} \times 2} = \frac{3.0\text{Ghz}}{30} = 0.1 \times 10^9$$

$$\begin{aligned}\text{Cache cycles available} &= \text{Memory references/clock/processor} \times \text{Clock rate} \\ &\quad \times \text{processor count} \times \text{CMR} \\ &= \frac{0.4}{0.7} \times 3.0\text{GHz} \times 8 \times \text{CMR} = 13.7 \times 10^9 \times \text{CMR}\end{aligned}$$

$$\text{CMR} = \frac{0.1}{13.7} = 0.0073 = 0.73\%$$

This means that the coherence miss rate must be 0.73% or less. In the next section, we will see several applications with coherence miss rates in excess of 1%. Alternatively, if we assume that CMR can be 1%, then we could support just under 6 processors. Clearly, even small multicores will require a method for scaling snoop bandwidth.

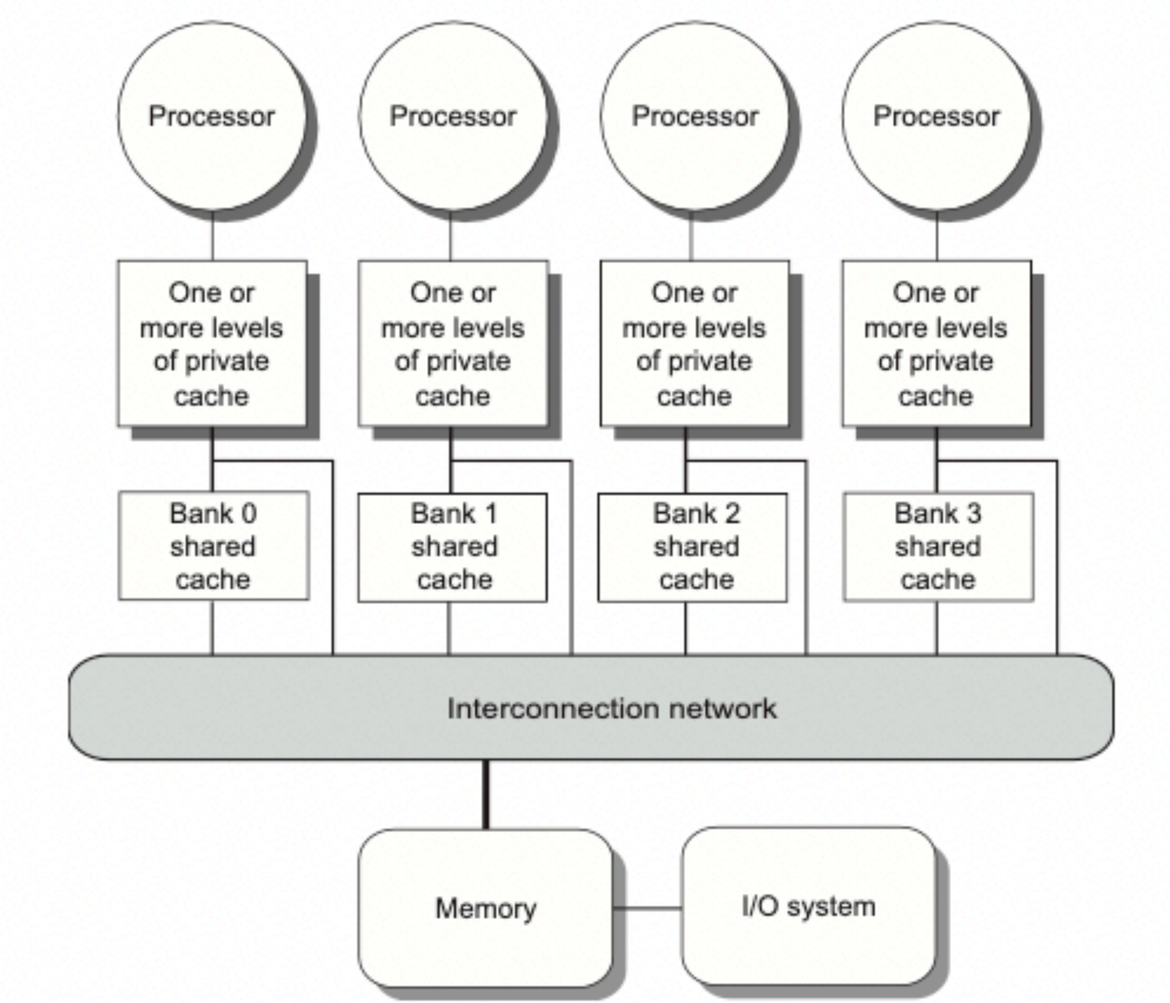

Figure 5.8 A single-chip multicore with a distributed cache. In current designs, the

NUCA - Non-uniform cache access

Example Assume that words z1 and z2 are in the same cache block, which is in the shared state in the caches of both P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss.

Time	P1	P2
1	Write z1	
2		Read z2
3	Write z1	
4		Write z2
5	Read z2	

Answer Here are the classifications by time step:

1. This event is a true sharing miss, since z1 is in the shared state in P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since z2 was invalidated by the write of z1 in P1, but that value of z1 is not used in P2.
3. This event is a false sharing miss, since the block containing z1 is marked shared due to the read in P2, but P2 did not read z1. The cache block containing z1 will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols, this will be handled as an *upgrade request*, which generates a bus invalidate, but does not transfer the cache block.
4. This event is a false sharing miss for the same reason as step 3.
5. This event is a true sharing miss, since the value being read was written by P2.

Cache level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I/8 KB D	32 KB I/32 KB D
	Associativity	Direct-mapped	8-way I/8-way D
	Block size	32 B	64 B
	Miss penalty	7	10
L2	Size	96 KB	256 KB
	Associativity	3-way	8-way
	Block size	32 B	64 B
	Miss penalty	21	35
L3	Size	2 MiB (total 8 MiB unshared)	2 MiB per core (8 MiB total shared)
	Associativity	Direct-mapped	16-way
	Block size	64 B	64 B
	Miss penalty	80	~100

Figure 5.9 The characteristics of the cache hierarchy of the Alpha 21164 used in this study and the Intel i7.

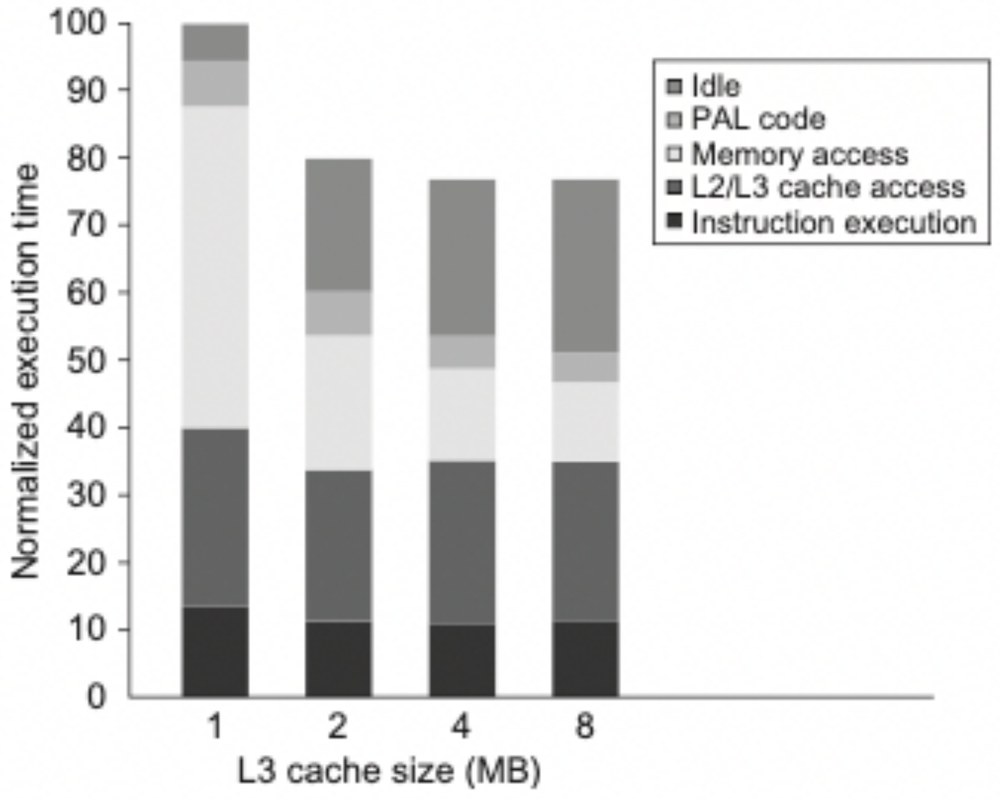

Figure 5.10 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 to 8 MiB. The idle time

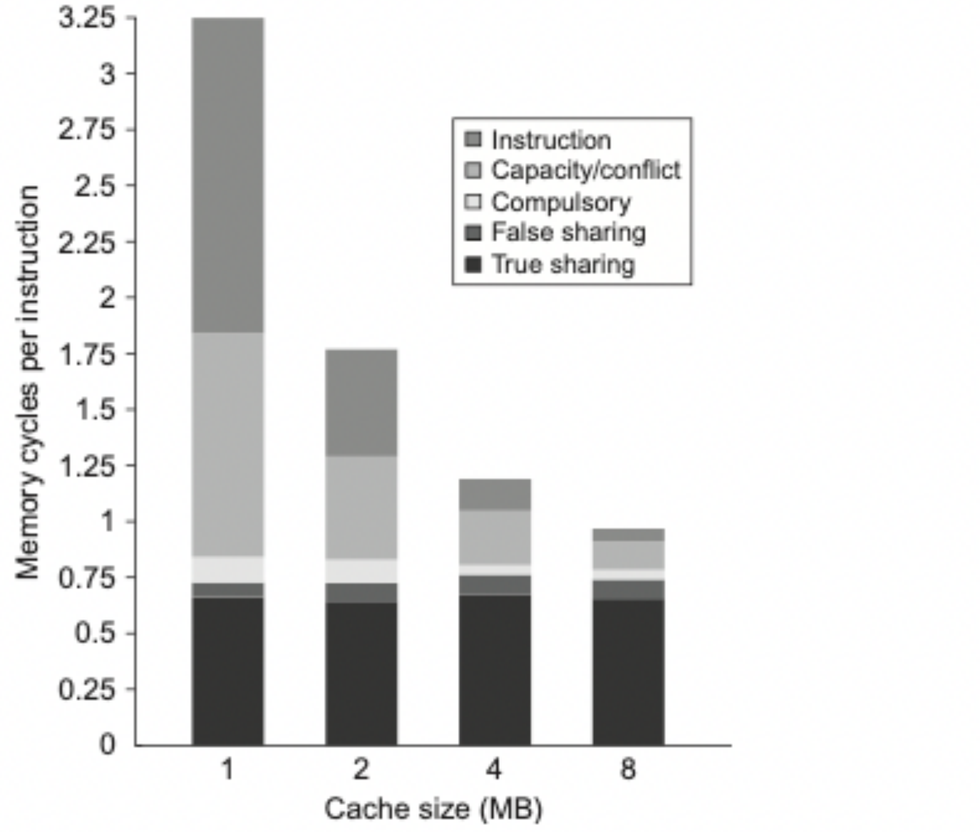

Figure 5.11 The contributing causes of memory access cycle shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

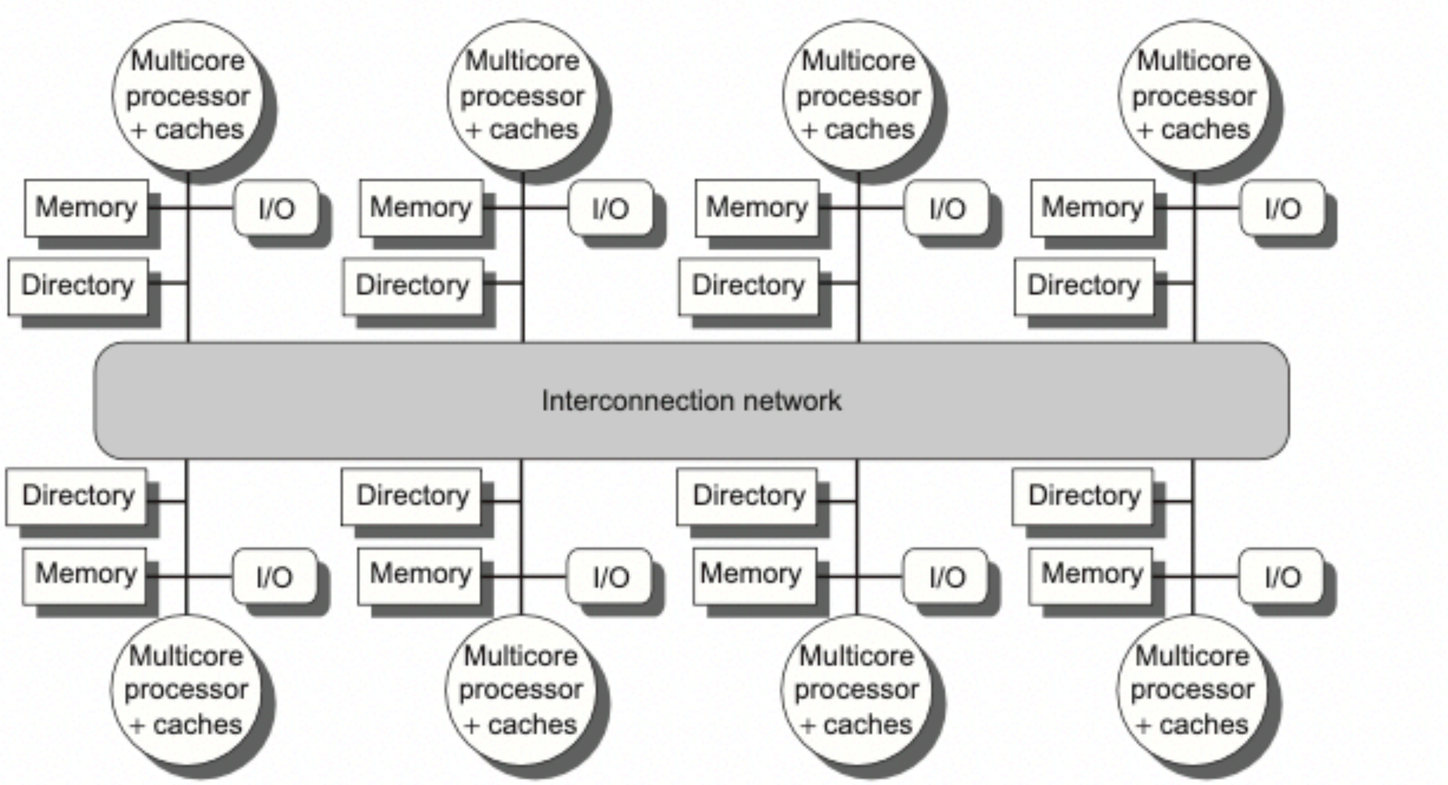

Figure 5.18 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. In this case, a node is shown as a single multicore chip, and the directory information for the associated memory may reside either on or off the multicore. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism will handle both the maintenance of the directory information and any coherence actions needed within the multicore node.

Directory-Based Coherence Protocols

- Shared - One or more nodes have copy, up-to-date
- Uncached - No node has a copy in cache
- Modified - one cache has written a value, other copies (including main memory) are out-of-date

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
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 5.19 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local node to the home. The

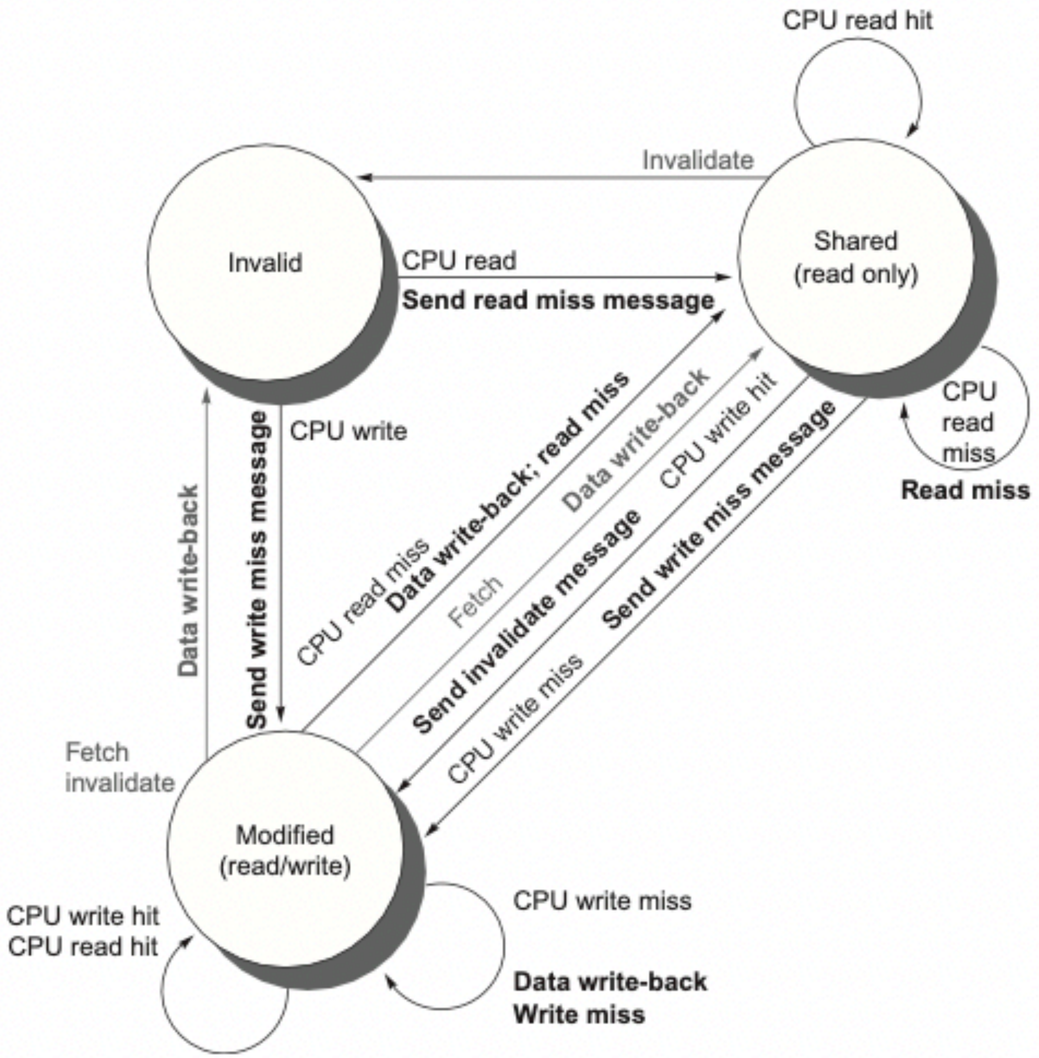

Figure 5.20 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in *black*, and those from the home directory are shown in *gray*. The states are identical to

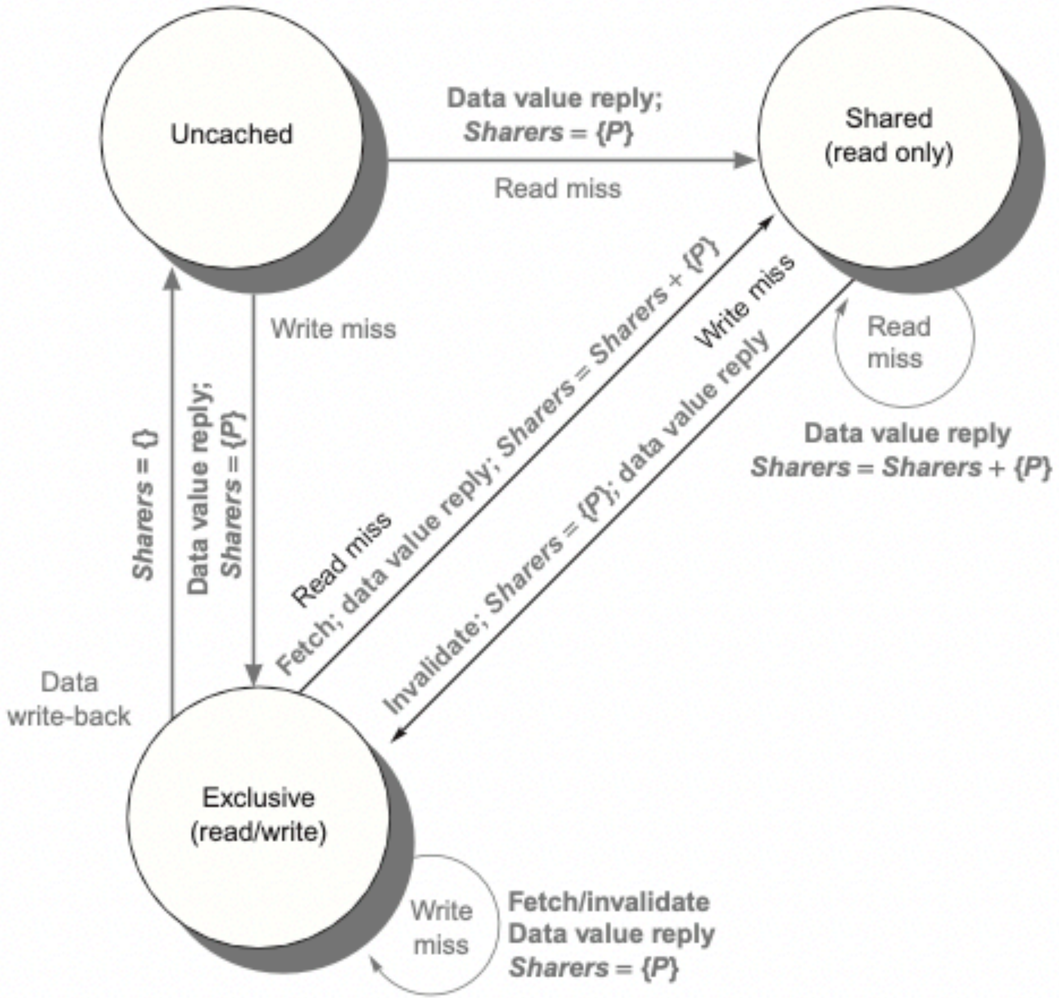

Figure 5.21 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. *Bold* indicates the action taken by the directory in response to the request.

Uncached and shared state behavior

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

Exclusive state behavior

- *Read miss*—The owner 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 node 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 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 node, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.