# Chapter 5 Solutions

## Case Study 1: Single-Chip Multicore Multiprocessor

5.1
a. P0: read 120 → P0.B0: (S, 120, 0020) returns 0020
b. P0: write 120 ← 80 → P0.B0: (M, 120, 0080)
P3.B0: (I, 120, 0020)
c. P3: write 120 ← 80 → P3.B0: (M, 120, 0080)
d. P1: read 110 → P1.B2: (S, 110, 0010) returns 0010
e. P0: write 108 ← 48 → P0.B1: (M, 108, 0048)
P3.B1: (I, 108, 0008)
f. P0: write 130 ← 78 → P0.B2: (M, 130, 0078)
M: 110 ← 0030 (writeback to memory)
g. P3: write 130 ← 78 → P3.B2: (M, 130, 0078)

5.2
a. P0: read 120, Read miss, satisfied by memory
P0: read 128, Read miss, satisfied by P1's cache
P0: read 130, Read miss, satisfied by memory, writeback 110
Implementation 1: 100 + 40 + 10 + 100 + 10 = 260 stall cycles
Implementation 2: 100 + 130 + 10 + 100 + 10 = 350 stall cycles
b. P0: read 100, Read miss, satisfied by memory
P0: write 108 ← 48, Write hit, sends invalidate
P0: write 130 ← 78, Write miss, satisfied by memory, write back 110
Implementation 1: 100 + 15 + 10 + 100 = 225 stall cycles
Implementation 2: 100 + 15 + 10 + 100 = 225 stall cycles
c. P1: read 120, Read miss, satisfied by memory
P1: read 128, Read hit
P1: read 130, Read miss, satisfied by memory
Implementation 1: 100 + 0 + 100 = 200 stall cycles
Implementation 2: 100 + 0 + 100 = 200 stall cycles
d. P1: read 100, Read miss, satisfied by memory
P1: write 108 ← 48, Write miss, satisfied by memory, write back 128
P1: write 130 ← 78, Write miss, satisfied by memory
Implementation 1: 100 + 100 + 10 + 100 = 310 stall cycles
Implementation 2: 100 + 100 + 10 + 100 = 310 stall cycles

5.3 See Figure S.28

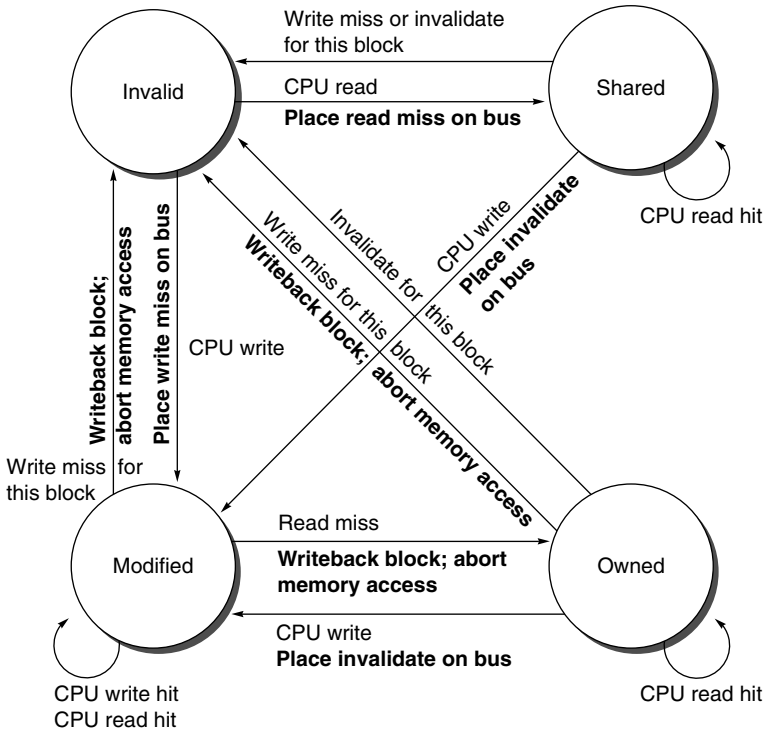


**Figure S.28 Protocol diagram.**

5.4 (Showing results for implementation 1)

a. P1: read 110, Read miss, P0's cache

P3: read 110, Read miss, MSI satisfies in memory, MOSI satisfies in P0's cache

P0: read 110, Read hit

MSI: 40 + 10 + 100 + 0 = 150 stall cycles

MOSI: 40 + 10 + 40 + 10 + 0 = 100 stall cycles

b. P1: read 120, Read miss, satisfied in memory

P3: read 120, Read hit

P0: read 120, Read miss, satisfied in memory

Both protocols: 100 + 0 + 100 = 200 stall cycles

c. P0: write 120 ← 80, Write miss, invalidates P3

P3: read 120, Read miss, P0's cache

P0: read 120, Read hit

Both protocols: 100 + 40 + 10 + 0 = 150 stall cycles

d. P0: write 108 ← 88, Send invalidate, invalidate P3
P3: read 108, Read miss, P0's cache
P0: write 108 ← 98, Send invalidate, invalidate P3
Both protocols: 15 + 40 + 10 + 15 = 80 stall cycles

5.5 See Figure S.29

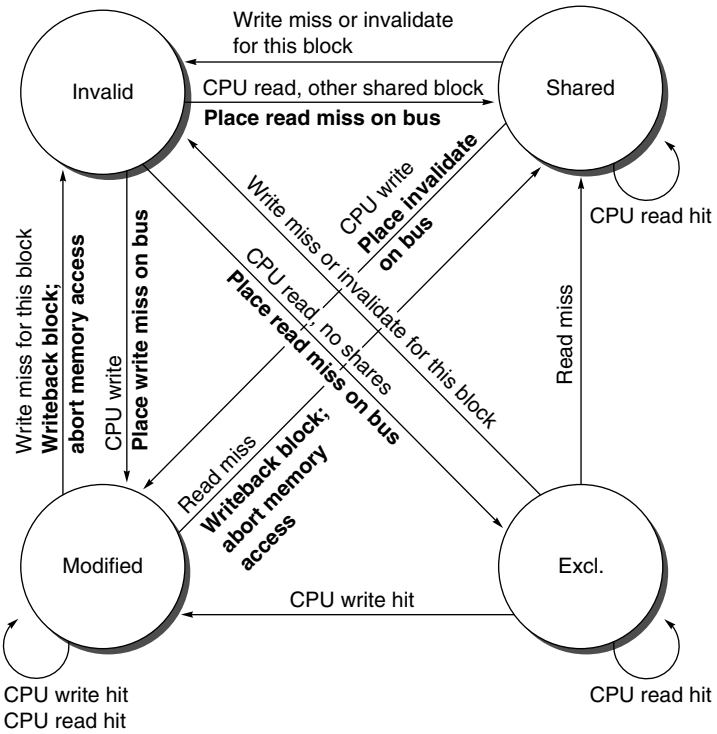


**Figure S.29 Diagram for a MESI protocol.**

5.6 a. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
p0: write 100 ← 40, MSI: send invalidate, MESI: silent transition from E to M
MSI: 100 + 15 = 115 stall cycles
MESI: 100 + 0 = 100 stall cycles

b. p0: read 120, Read miss, satisfied in memory, sharers both to S
p0: write 120 ← 60, Both send invalidates
Both: 100 + 15 = 115 stall cycles

c. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
p0: read 120, Read miss, memory, silently replace 120 from S or E
Both: 100 + 100 = 200 stall cycles, silent replacement from E

d. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E

p1: write 100 ← 60, Write miss, satisfied in memory regardless of protocol

Both: 100 + 100 = 200 stall cycles, don't supply data in E state (some protocols do)

e. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E

p0: write 100 ← 60, MSI: send invalidate, MESI: silent transition from E to M

p1: write 100 ← 40, Write miss, P0's cache, writeback data to memory

MSI: 100 + 15 + 40 + 10 = 165 stall cycles

MESI: 100 + 0 + 40 + 10 = 150 stall cycles

5.7 a. Assume the processors acquire the lock in order. P0 will acquire it first, incurring 100 stall cycles to retrieve the block from memory. P1 and P3 will stall until P0's critical section ends (ping-ponging the block back and forth) 1000 cycles later. P0 will stall for (about) 40 cycles while it fetches the block to invalidate it; then P1 takes 40 cycles to acquire it. P1's critical section is 1000 cycles, plus 40 to handle the write miss at release. Finally, P3 grabs the block for a final 40 cycles of stall. So, P0 stalls for 100 cycles to acquire, 10 to give it to P1, 40 to release the lock, and a final 10 to hand it off to P1, for a total of 160 stall cycles. P1 essentially stalls until P0 releases the lock, which will be 100 + 1000 + 10 + 40 = 1150 cycles, plus 40 to get the lock, 10 to give it to P3, 40 to get it back to release the lock, and a final 10 to hand it back to P3. This is a total of 1250 stall cycles. P3 stalls until P1 hands it off the released lock, which will be 1150 + 40 + 10 + 1000 + 40 = 2240 cycles. Finally, P3 gets the lock 40 cycles later, so it stalls a total of 2280 cycles.

b. The optimized spin lock will have many fewer stall cycles than the regular spin lock because it spends most of the critical section sitting in a spin loop (which while useless, is not defined as a stall cycle). Using the analysis below for the interconnect transactions, the stall cycles will be 3 read memory misses (300), 1 upgrade (15) and 1 write miss to a cache (40 + 10) and 1 write miss to memory (100), 1 read cache miss to cache (40 + 10), 1 write miss to memory (100), 1 read miss to cache and 1 read miss to memory (40 + 10 + 100), followed by an upgrade (15) and a write miss to cache (40 + 10), and finally a write miss to cache (40 + 10) followed by a read miss to cache (40 + 10) and an upgrade (15). So approximately 945 cycles total.

c. Approximately 31 interconnect transactions. The first processor to win arbitration for the interconnect gets the block on its first try (1); the other two ping-pong the block back and forth during the critical section. Because the latency is 40 cycles, this will occur about 25 times (25). The first processor does a write to release the lock, causing another bus transaction (1), and the second processor does a transaction to perform its test and set (1). The last processor gets the block (1) and spins on it until the second processor releases it (1). Finally the last processor grabs the block (1).

d. Approximately 15 interconnect transactions. Assume processors acquire the lock in order. All three processors do a test, causing a read miss, then a test and set, causing the first processor to upgrade and the other two to write miss (6). The losers sit in the test loop, and one of them needs to get back a shared block first (1). When the first processor releases the lock, it takes a write miss (1) and then the two losers take read misses (2). Both have their test succeed, so the new winner does an upgrade and the new loser takes a write miss (2). The loser spins on an exclusive block until the winner releases the lock (1). The loser first tests the block (1) and then test-and-sets it, which requires an upgrade (1).

5.8 Latencies in implementation 1 of Figure 5.36 are used.

| | | |
|---|---|---|
| a. | P0: write 110 ← 80 | Hit in P0's cache, no stall cycles for either TSO or SC |
| | P0: read 108 | Hit in P0's cache, no stall cycles for either TSO or SC |
| b. | P0: write 100 ← 80 | Miss, TSO satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles) |
| | P0: read 108 | Hit, but must wait for preceding operation: TSO = 0, SC = 100 |
| c. | P0: write 110 ← 80 | Hit in P0's cache, no stall cycles for either TSO or SC |
| | P0: write 100 ← 90 | Miss, TSO satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles) |
| d. | P0: write 100 ← 80 | Miss, TSO satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles) |
| | P0: write 110 ← 90 | Hit, but must wait for preceding operation: TSO = 0, SC = 100 |

## Case Study 2: Simple Directory-Based Coherence

| | | |
|---|---|---|
| 5.9 a. | P0,0: read 100 | L1 hit returns 0x0010, state unchanged (M) |
| b. | P0,0: read 128 | L1 miss and L2 miss will replace B1 in L1 and B1 in L2 which has address 108. |
| | | L1 will have 128 in B1 (shared), L2 also will have it (DS, P0,0) |
| | | Memory directory entry for 108 will become <DS, C1> |
| | | Memory directory entry for 128 will become <DS, C0> |

c, d, …, h: follow same approach

5.10 a. P0,0: write 100 ← 80, Write hit only seen by P0,0

b. P0,0: write 108 ← 88, Write "upgrade" received by P0,0; invalidate received by P3,1

c. P0,0: write 118 ← 90, Write miss received by P0,0; invalidate received by P1,0

d. P1,0: write 128 ← 98, Write miss received by P1,0.

5.11 a. See Figures S.30 and S.31

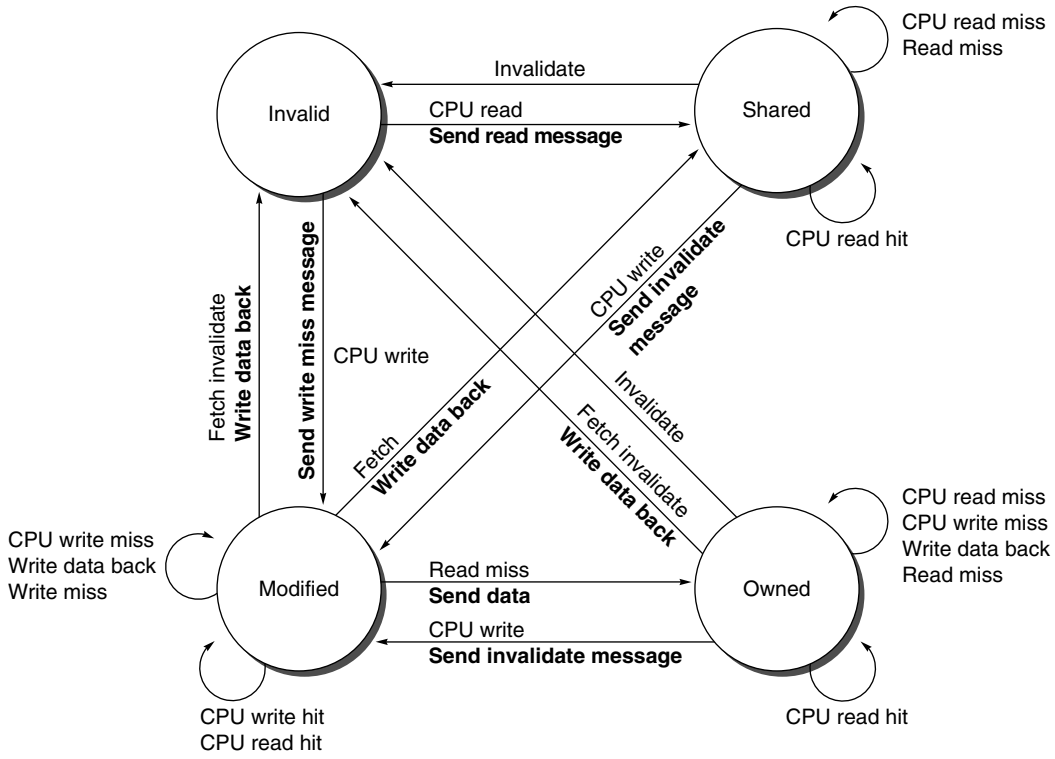


**Figure S.30 Cache states.**

5.12 The Exclusive state (E) combines properties of Modified (M) and Shared (S). The E state allows silent upgrades to M, allowing the processor to write the block without communicating this fact to memory. It also allows silent downgrades to I, allowing the processor to discard its copy with notifying memory. The memory must have a way of inferring either of these transitions. In a directory-based system, this is typically done by having the directory assume that the node is in state M and forwarding all misses to that node. If a node has silently downgraded to I, then it sends a NACK (Negative Acknowledgment) back to the directory, which then infers that the downgrade occurred. However, this results in a race with other messages, which can cause other problems.

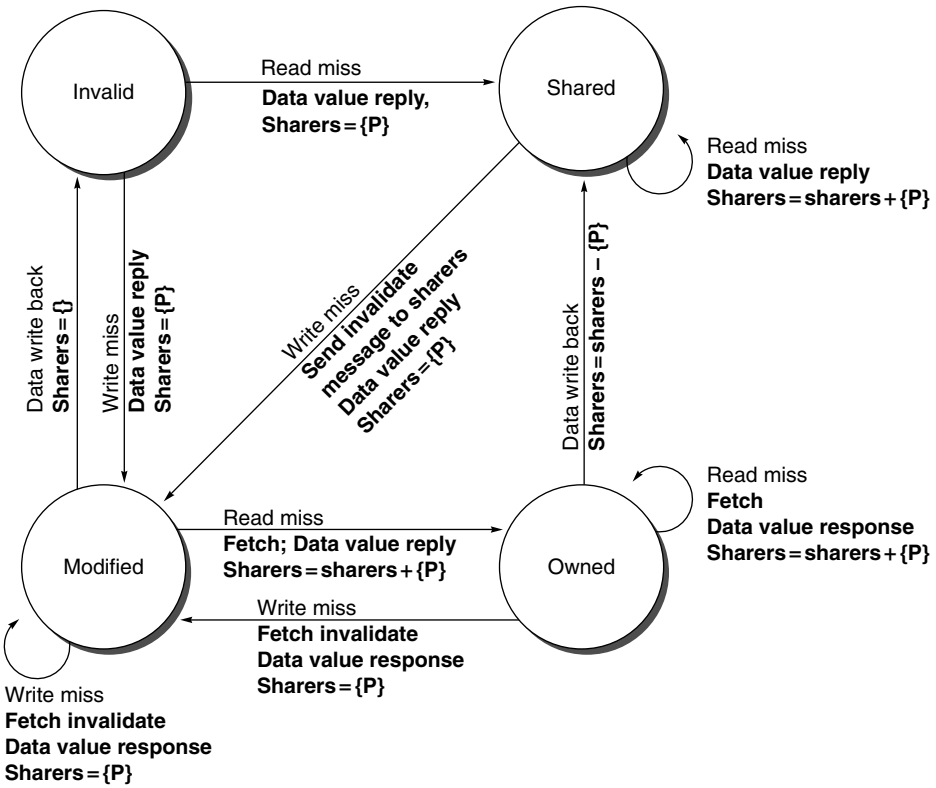


**Figure S.31** Directory states.

## Case Study 3: Advanced Directory Protocol

5.13 a. P0,0: read 100 — Read hit

b. P0,0: read 120 — Miss, will replace modified data (B0) and get new line in shared state

P0,0: M → $MI^A$ → I → $IS^D$ → S  Dir: DM {P0,0} → DI {}

c. P0,0: write 120 ← 80 — Miss will replace modified data (B0) and get new line in modified state

P0,0: M → $MI^A$ → I → $IM^{AD}$ → $IM^A$ → M

P3,1: S → I

Dir: DS {P3,0} → DM {P0,0}

d, e, f: steps similar to parts a, b, and c

5.14 a. P0,0: read 120 — Miss, will replace modified data (B0) and get new line in shared state

P0,0: M → $MI^A$ → I → $IS^D$ → S

| | | |
|---|---|---|
| | P1,0: read 120 | Miss, will replace modified data (B0) and get new line in shared state<br>P1,0: M → $MI^A$ → I → $IS^D$ → S<br>Dir: DS {P3,0} → DS {P3,0; P0,0} → DS {P3,0; P0,0; P1,0} |
| b. | P0,0: read 120 | Miss, will replace modified data (B0) and get new line in shared state<br>P0,0: M → $MI^A$ → I → $IS^D$ → S |
| | P1,0: write 120 ← 80 | Miss will replace modified data (B0) and get new line in modified state<br>P1,0: M → $MI^A$ → I → $IM^{AD}$ → $IM^A$ → M<br>P3,1: S → I<br>Dir: DS {P3,1} → DS {P3,0; P1,0} → DM {P1,0} |
| | c, d, e: steps similar to parts a and b | |

5.15

| | | |
|---|---|---|
| a. | P0,0: read 100 | Read hit, 1 cycle |
| b. | P0,0: read 120 | Read Miss, causes modified block replacement and is satisfied in memory and incurs 4 chip crossings (see underlined) |
| | Latency for P0,0: | Lsend_data + Ldata_msg + Lwrite_memory + Linv + L_ack + Lreq_msg + Lsend_msg + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data + 4 × chip crossings latency = 20 + 30 + 20 + 1 + 4 + 15 + 6 + 15 + 100 + 30 + 15 + 4 × 20 = 336 |
| | c, d, e: follow same steps as a and b | |

5.16 All protocols must ensure forward progress, even under worst-case memory access patterns. It is crucial that the protocol implementation guarantee (at least with a probabilistic argument) that a processor will be able to perform at least one memory operation each time it completes a cache miss. Otherwise, starvation might result. Consider the simple spin lock code:

```
tas:
DADDUI R2, R0, #1
lockit:
EXCH R2, 0(R1)
BNEZ R2, lockit
```

If all processors are spinning on the same loop, they will all repeatedly issue GetM messages. If a processor is not guaranteed to be able to perform at least one instruction, then each could steal the block from the other repeatedly. In the worst case, no processor could ever successfully perform the exchange.

5.17 a. The $MS^A$ state is essentially a "transient O" because it allows the processor to read the data and it will respond to GetShared and GetModified requests from other processors. It is transient, and not a real O state, because memory will send the PutM_Ack and take responsibility for future requests.

b. See Figures S.32 and S.33

| State | Read | Write | Replace-ment | INV | Forwarded_ GetS | Forwarded_ GetM | PutM_ Ack | Data | Last ACK |
|---|---|---|---|---|---|---|---|---|---|
| I | send GetS/IS | send GetM/IM | error | send Ack/I | error | error | error | error | error |
| S | do Read | send GetM/IM | I | send Ack/I | error | error | error | error | error |
| O | do Read | send GetM/OM | send PutM/OI | error | send Data | send Data/I | error | — | — |
| M | do Read | do Write | send PutM/MI | error | send Data/O | send Data/I | error | error | error |
| IS | z | z | z | send Ack/ISI | error | error | error | save Data, do Read/S | error |
| ISI | z | z | z | send Ack | error | error | error | save Data, do Read/I | error |
| IM | z | z | z | send Ack | IMO | $IMI^A$ | error | save Data | do Write/M |
| IMI | z | z | z | error | error | error | error | save Data | do Write, send Data/I |
| IMO | z | z | z | send Ack/IMI | — | IMOI | error | save Data | do Write, send Data/O |
| IMOI | z | z | z | error | error | error | error | save Data | do Write, send Data/I |
| OI | z | z | z | error | send Data | send Data | /I | error | error |
| MI | z | z | z | error | send Data | send Data | /I | error | error |
| OM | z | z | z | error | send Data | send Data/IM | error | save Data | do Write/M |

**Figure S.32 Directory protocol cache controller transitions.**

| State | Read | Write | Replacement (owner) | INV (nonowner) |
|---|---|---|---|---|
| DI | send Data, add to sharers/DS | send Data, clear sharers, set owner/DM | error | send PutM_Ack |
| DS | send Data, add to sharers | send INVs to sharers, clear sharers, set owner, send Data/DM | error | send PutM_Ack |
| DO | forward GetS, add to sharers | forward GetM, send INVs to sharers, clear sharers, set owner/DM | send Data, send PutM_Ack/DS | send PutM_Ack |
| DM | forward GetS, add to requester and owner to sharers/DO | forward GetM, send INVs to sharers, clear sharers, set owner | send Data, send PutM_Ack/DI | send PutM_Ack |

**Figure S.33 Directory controller transitions.**

5.18 a. P1,0: read 100

P3,1: write 100 ← 90

In this problem, both P0,1 and P3,1 miss and send requests that race to the directory. Assuming that P0,1's GetS request arrives first, the directory will forward P0,1's GetS to P0,0, followed shortly afterwards by P3,1's GetM. If the network maintains point-to-point order, then P0,0 will see the requests in the right order and the protocol will work as expected. However, if the forwarded requests arrive out of order, then the GetX will force P0 to state I, causing it to detect an error when P1's forwarded GetS arrives.

b. P1,0: read 100

P0,0: replace 100

P1,0's GetS arrives at the directory and is forwarded to P0,0 before P0,0's PutM message arrives at the directory and sends the PutM_Ack. However, if the PutM_Ack arrives at P0,0 out of order (i.e., before the forwarded GetS), then this will cause P0,0 to transition to state I. In this case, the forwarded GetS will be treated as an error condition.

## Exercises

5.19 The general form for Amdahl's Law is

$$Speedup = \frac{Execution\ time_{old}}{Execution\ time_{new}}$$

all that needs to be done to compute the formula for speedup in this multiprocessor case is to derive the new execution time.

The exercise states that for the portion of the original execution time that can use $i$ processors is given by $F(i,p)$. If we let $Execution\ time_{old}$ be 1, then the relative time for the application on $p$ processors is given by summing the times required for each portion of the execution time that can be sped up using $i$ processors, where $i$ is between 1 and $p$. This yields

$$Execution\ time_{new} = \sum_{i=1}^{p} \frac{f(i,p)}{i}$$

Substituting this value for $Execution\ time_{new}$ into the speedup equation makes Amdahl's Law a function of the available processors, $p$.

5.20 a. (i) 64 processors arranged a as a ring: largest number of communication hops = 32 → communication cost = (100 + 10 × 32) ns = 420 ns.

(ii) 64 processors arranged as 8x8 processor grid: largest number of communication hops = 14 → communication cost = (100 + 10 × 14) ns = 240 ns.

(iii) 64 processors arranged as a hypercube: largest number of hops = 6 ($\log_2$ 64) → communication cost = (100 + 10 × 6) ns = 160 ns.

b. Base CPI = 0.5 *cpi*

(i) 64 processors arranged a as a ring: Worst case CPI = 0.5 + 0.2/100 × (420) = 1.34 *cpi*

(ii) 64 processors arranged as 8x8 processor grid: Worst case CPI = 0.5 + 0.2/100 × (240) = 0.98 *cpi*

(iii) 64 processors arranged as a hypercube: Worst case CPI CPI = 0.5 + 0.2/100 × (160) = 0.82 *cpi*

The average CPI can be obtained by replacing the largest number of communications hops in the above calculation by $\hat{h}$, the average numbers of communications hops. That latter number depends on both the topology and the application.

c. Since the CPU frequency and the number of instructions executed did not change, the answer can be obtained by the CPI for each of the topologies (worst case or average) by the base (no remote communication) CPI.

5.21 To keep the figures from becoming cluttered, the coherence protocol is split into two parts as was done in Figure 5.6 in the text. Figure S.34 presents the CPU portion of the coherence protocol, and Figure S.35 presents the bus portion of the protocol. In both of these figures, the arcs indicate transitions and the text along each arc indicates the stimulus (in normal text) and bus action (in bold text) that occurs during the transition between states. Finally, like the text, we assume a write hit is handled as a write miss.

Figure S.34 presents the behavior of state transitions caused by the CPU itself. In this case, a write to a block in either the invalid or shared state causes us to broadcast a "write invalidate" to flush the block from any other caches that hold the block and move to the exclusive state. We can leave the exclusive state through either an invalidate from another processor (which occurs on the bus side of the coherence protocol state diagram), or a read miss generated by the CPU (which occurs when an exclusive block of data is displaced from the cache by a second block). In the shared state only a write by the CPU or an invalidate from another processor can move us out of this state. In the case of transitions caused by events external to the CPU, the state diagram is fairly simple, as shown in Figure S.35. When another processor writes a block that is resident in our cache, we unconditionally invalidate the corresponding block in our cache. This ensures that the next time we read the data, we will load the updated value of the block from memory. Also, whenever the bus sees a read miss, it must change the state of an exclusive block to shared as the block is no longer exclusive to a single cache.

The major change introduced in moving from a write-back to write-through cache is the elimination of the need to access dirty blocks in another processor's caches. As a result, in the write-through protocol it is no longer necessary to provide the hardware to force write back on read accesses or to abort pending memory accesses. As memory is updated during any write on a write-through cache, a processor that generates a read miss will always retrieve the correct information from memory. Basically, it is not possible for valid cache blocks to be incoherent with respect to main memory in a system with write-through caches.

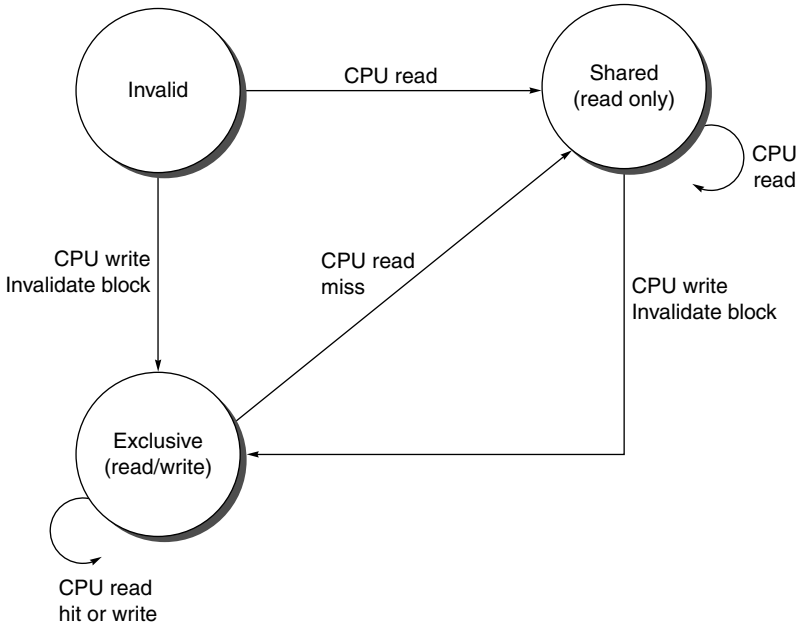


**Figure S.34** CPU portion of the simple cache coherency protocol for write-through caches.

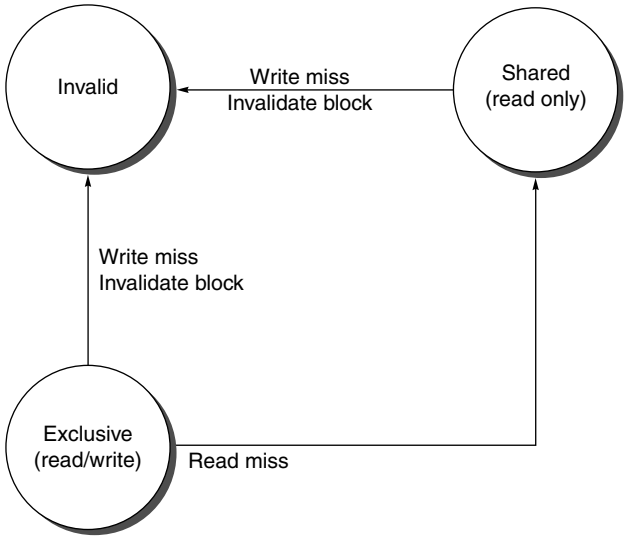


**Figure S.35** Bus portion of the simple cache coherency protocol for write-through caches.

5.22 To augment the snooping protocol of Figure 5.7 with a Clean Exclusive state we assume that the cache can distinguish a read miss that will allocate a block destined to have the Clean Exclusive state from a read miss that will deliver a Shared block. Without further discussion we assume that there is some mechanism to do so.

The three states of Figure 5.7 and the transitions between them are unchanged, with the possible clarifying exception of renaming the Exclusive (read/write) state to Dirty Exclusive (read/write).

The new Clean Exclusive (read only) state should be added to the diagram along with the following transitions.

- from Clean Exclusive to Clean Exclusive in the event of a CPU read hit on this block or a CPU read miss on a Dirty Exclusive block
- from Clean Exclusive to Shared in the event of a CPU read miss on a Shared block or on a Clean Exclusive block
- from Clean Exclusive to Shared in the event of a read miss on the bus for this block
- from Clean Exclusive to Invalid in the event of a write miss on the bus for this block
- from Clean Exclusive to Dirty Exclusive in the event of a CPU write hit on this block or a CPU write miss
- from Dirty Exclusive to Clean Exclusive in the event of a CPU read miss on a Dirty Exclusive block
- from Invalid to Clean Exclusive in the event of a CPU read miss on a Dirty Exclusive block
- from Shared to Clean Exclusive in the event of a CPU read miss on a Dirty Exclusive block

Several transitions from the original protocol must change to accommodate the existence of the Clean Exclusive state. The following three transitions are those that change.

- from Dirty Exclusive to Shared, the label changes to CPU read miss on a Shared block
- from Invalid to Shared, the label changes to CPU miss on a Shared block
- from Shared to Shared, the miss transition label changes to CPU read miss on a Shared block

5.23 An obvious complication introduced by providing a valid bit per word is the need to match not only the tag of the block but also the offset within the block when snooping the bus. This is easy, involving just looking at a few more bits. In addition, however, the cache must be changed to support write-back of partial cache blocks. When writing back a block, only those words that are valid should be written to memory because the contents of invalid words are not necessarily coherent

with the system. Finally, given that the state machine of Figure 5.7 is applied at each cache block, there must be a way to allow this diagram to apply when state can be different from word to word within a block. The easiest way to do this would be to provide the state information of the figure for each word in the block. Doing so would require much more than one valid bit per word, though. Without replication of state information the only solution is to change the coherence protocol slightly.

5.24 a. The instruction execution component would be significantly sped up because the out-of-order execution and multiple instruction issue allows the latency of this component to be overlapped. The cache access component would be similarly sped up due to overlap with other instructions, but since cache accesses take longer than functional unit latencies, they would need more instructions to be issued in parallel to overlap their entire latency. So the speedup for this component would be lower.

The memory access time component would also be improved, but the speedup here would be lower than the previous two cases. Because the memory comprises local and remote memory accesses and possibly other cache-to-cache transfers, the latencies of these operations are likely to be very high (100's of processor cycles). The 64-entry instruction window in this example is not likely to allow enough instructions to overlap with such long latencies. There is, however, one case when large latencies can be overlapped: when they are hidden under other long latency operations. This leads to a technique called miss-clustering that has been the subject of some compiler optimizations. The other-stall component would generally be improved because they mainly consist of resource stalls, branch mispredictions, and the like. The synchronization component if any will not be sped up much.

b. Memory stall time and instruction miss stall time dominate the execution for OLTP, more so than for the other benchmarks. Both of these components are not very well addressed by out-of-order execution. Hence the OLTP workload has lower speedup compared to the other benchmarks with System B.

5.25 Because false sharing occurs when both the data object size is smaller than the granularity of cache block valid bit(s) coverage and more than one data object is stored in the same cache block frame in memory, there are two ways to prevent false sharing. Changing the cache block size or the amount of the cache block covered by a given valid bit are hardware changes and outside the scope of this exercise. However, the allocation of memory locations to data objects is a software issue.

The goal is to locate data objects so that only one truly shared object occurs per cache block frame in memory and that no non-shared objects are located in the same cache block frame as any shared object. If this is done, then even with just a single valid bit per cache block, false sharing is impossible. Note that shared, read-only-access objects could be combined in a single cache block and not contribute to the false sharing problem because such a cache block can be held by many caches and accessed as needed without an invalidations to cause unnecessary cache misses.

To the extent that shared data objects are explicitly identified in the program source code, then the compiler should, with knowledge of memory hierarchy details, be able to avoid placing more than one such object in a cache block frame in memory. If shared objects are not declared, then programmer directives may need to be added to the program. The remainder of the cache block frame should not contain data that would cause false sharing misses. The sure solution is to pad with block with non-referenced locations.

Padding a cache block frame containing a shared data object with unused memory locations may lead to rather inefficient use of memory space. A cache block may contain a shared object plus objects that are read-only as a trade-off between memory use efficiency and incurring some false-sharing misses. This optimization almost certainly requires programmer analysis to determine if it would be worthwhile. Generally, careful attention to data distribution with respect to cache lines and partitioning the computation across processors is needed.

5.26 The problem illustrates the complexity of cache coherence protocols. In this case, this could mean that the processor P1 evicted that cache block from its cache and immediately requested the block in subsequent instructions. Given that the write-back message is longer than the request message, with networks that allow out-of-order requests, the new request can arrive before the write back arrives at the directory. One solution to this problem would be to have the directory wait for the write back and then respond to the request. Alternatively, the directory can send out a negative acknowledgment (NACK). Note that these solutions need to be thought out very carefully since they have potential to lead to deadlocks based on the particular implementation details of the system. Formal methods are often used to check for races and deadlocks.

5.27 If replacement hints are used, then the CPU replacing a block would send a hint to the home directory of the replaced block. Such hint would lead the home directory to remove the CPU from the sharing list for the block. That would save an invalidate message when the block is to be written by some other CPU. Note that while the replacement hint might reduce the total protocol latency incurred when writing a block, it does not reduce the protocol traffic (hints consume as much bandwidth as invalidates).

5.28 a. Considering first the storage requirements for nodes that are caches under the directory subtree:

The directory at any level will have to allocate entries for all the cache blocks cached under that directory's subtree. In the worst case (all the CPU's under the subtree are not sharing any blocks), the directory will have to store as many entries as the number of blocks of all the caches covered in the subtree. That means that the root directory might have to allocate enough entries to reference all the blocks of all the caches. Every memory block cached in a directory will represented by an entry <block address, k-bit vector>, the k-bit vector will have a bit specifying all the subtrees that have a copy of the block. For example, for a binary tree an entry <m, 11> means that block m is cached under both branches of the tree. To be more precise, one bit per subtree would

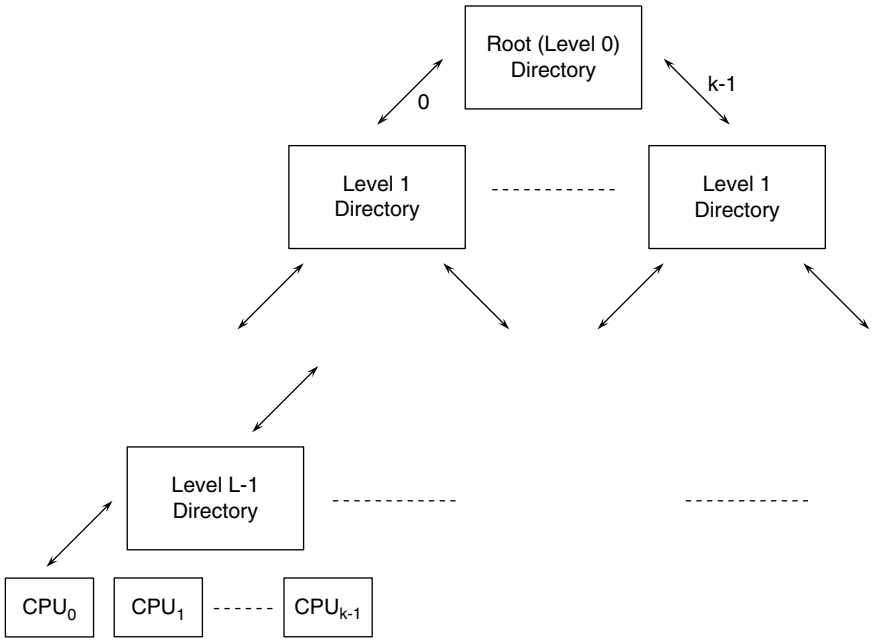


**Figure S.36** **Tree-based directory hierarchy (k-ary tree with l levels).**

be adequate if only the valid/invalid states need to be recorded; however to record whether a block is modified or not, more bits would be needed. Note that no entry is needed if a block is not cached under the subtree.

If the cache block has m bits (tag + index) then and s state bits need to be stored per block, and the cache can hold b blocks, then the directories at level L-1 (lowest level just above CPU's) will have to hold $k \times b$ entries. Each entry will have $(m + k \times s)$ bits. Thus each directory at level L-1 will have $(mkb + k^2bs)$ bits. At the next level of the hierarchy, the directories will be k times bigger. The number of directories at level i is $k^i$.

To consider memory blocks with a home in the subtree cached outside the subtree. The storage requirements per directory would have to be modified. Calculation outline:

Note that for some directory (for example the ones at level l-1) the number of possible home nodes that can be cached outside the subtree is equal to $(b \times ( k^l - x))$, where $k^l$ is the total number of CPU's, b is the number of blocks per cache and x is the number of CPU's under the directory's subtree. It should be noted that the extra storage diminishes for directories in higher levels of the tree (for example the directory at level 0 does not require any such storage since all the blocks have a home in that directory's subtree).

b. Simulation.

5.29 Test and set code using load linked and store conditional.

```
MOV R3, #1
LL  R2, 0(R1)
SC R3,  0(R1)
```

Typically this code would be put in a loop that spins until a 1 is returned in R3.

5.30 Assume a cache line that has a synchronization variable and the data guarded by that synchronization variable in the same cache line. Assume a two processor system with one processor performing multiple writes on the data and the other processor spinning on the synchronization variable. With an invalidate protocol, false sharing will mean that every access to the cache line ends up being a miss resulting in significant performance penalties.

5.31 The monitor has to be place at a point through which all memory accesses pass. One suitable place will be in the memory controller at some point where accesses from the 4 cores converge (since the accesses are uncached anyways). The monitor will use some sort of a cache where the tag of each valid entry is the address accessed by some load-linked instruction. In the data field of the entry, the core number that produced the load-linked access -whose address is stored in the tag field- is stored.

This is how the monitor reacts to the different memory accesses.

- Read not originating from a load-linked instruction:
  - Bypasses the monitor progresses to read data from memory
- Read originating from a load-linked instruction:
  - Checks the cache, if there is any entry with whose address matches the read address even if there is a partial address match (for example, read [0:7] and read [4:11] overlap match in addresses [4:7]), the matching cache entry is invalidated and a new entry is created for the new read (recording the core number that it belongs to). If there is no matching entry in the cache, then a new entry is created (if there is space in the cache). In either case the read progresses to memory and returns data to originating core.
- Write not originating from a store-conditional instruction:
  - Checks the cache, if there is any entry with whose address matches the write address even if there is a partial address match (for example, read [0:7] and write [4:11] overlap match in addresses [4:7]), the matching cache entry is invalidated. The write progresses to memory and writes data to the intended address.
- Write originating from a store-conditional instruction:
  - Checks the cache, if there is any entry with whose address matches the write address even if there is a partial address match (for example, read [0:7] and write [4:11] overlap match in addresses [4:7]), the core number in the cache entry is compared to the core that originated the write.

If the core numbers are the same, then the matching cache entry is invalidated, the write proceeds to memory and returns a success signal to the originating core. In that case, we expect the address match to be perfect – not partial- as we expect that the same core will not issue load-linked/store conditional instruction pairs that have overlapping address ranges.

If the core numbers differ, then the matching cache entry is invalidated, the write is aborted and returns a failure signal to the originating core. This case signifies that synchronization variable was corrupted by another core or by some regular store operation.

5.32 a. Because flag is written only after A is written, we would expect C to be 2000, the value of A.

b. Case 1: If the write to flag reached P2 faster than the write to A.

Case 2: If the read to A was faster than the read to flag.

c. Ensure that writes by P1 are carried out in program order and that memory operations execute atomically with respect to other memory operations.

To get intuitive results of sequential consistency using barrier instructions, a barrier need to be inserted in P1 between the write to A and the write to flag.

5.33 Inclusion states that each higher level of cache contains all the values present in the lower cache levels, i.e., if a block is in L1 then it is also in L2. The problem states that L2 has equal or higher associativity than L1, both use LRU, and both have the same block size.

When a miss is serviced from memory, the block is placed into all the caches, i.e., it is placed in L1 and L2. Also, a hit in L1 is recorded in L2 in terms of updating LRU information. Another key property of LRU is the following. Let A and B both be sets whose elements are ordered by their latest use. If A is a subset of B such that they share their most recently used elements, then the LRU element of B must either be the LRU element of A or not be an element of A.

This simply states that the LRU ordering is the same regardless if there are 10 entries or 100. Let us assume that we have a block, D, that is in L1, but not in L2. Since D initially had to be resident in L2, it must have been evicted. At the time of eviction D must have been the least recently used block. Since an L2 eviction took place, the processor must have requested a block not resident in L1 and obviously not in L2. The new block from memory was placed in L2 (causing the eviction) and placed in L1 causing yet another eviction. L1 would have picked the least recently used block to evict.

Since we know that D is in L1, it must be the LRU entry since it was the LRU entry in L2 by the argument made in the prior paragraph. This means that L1 would have had to pick D to evict. This results in D not being in L1 which results in a contradiction from what we assumed. If an element is in L1 it has to be in L2 (inclusion) given the problem's assumptions about the cache.

5.34 Analytical models can be used to derive high-level insight on the behavior of the system in a very short time. Typically, the biggest challenge is in determining the values of the parameters. In addition, while the results from an analytical model can give a good approximation of the relative trends to expect, there may be significant errors in the absolute predictions.

Trace-driven simulations typically have better accuracy than analytical models, but need greater time to produce results. The advantages are that this approach can be fairly accurate when focusing on specific components of the system (e.g., cache system, memory system, etc.). However, this method does not model the impact of aggressive processors (mispredicted path) and may not model the actual order of accesses with reordering. Traces can also be very large, often taking gigabytes of storage, and determining sufficient trace length for trustworthy results is important. It is also hard to generate representative traces from one class of machines that will be valid for all the classes of simulated machines. It is also harder to model synchronization on these systems without abstracting the synchronization in the traces to their high-level primitives.

Execution-driven simulation models all the system components in detail and is consequently the most accurate of the three approaches. However, its speed of simulation is much slower than that of the other models. In some cases, the extra detail may not be necessary for the particular design parameter of interest.

5.35 One way to devise a multiprocessor/cluster benchmark whose performance gets worse as processors are added:

Create the benchmark such that all processors update the same variable or small group of variables continually after very little computation.

For a multiprocessor, the miss rate and the continuous invalidates in between the accesses may contribute more to the execution time than the actual computation, and adding more CPU's could slow the overall execution time.

For a cluster organized as a ring communication costs needed to update the common variables could lead to inverse linear speedup behavior as more processors are added.