Chapter 5 Solutions

Case Study 1: Single-Chip Multicore Multiprocessor

5.1

5.1 Cx.y is cache line y in core x.

a. C0: R AC20 → C0.0: (S, AC20, 0020), returns 0020

b. C0: W AC20 ← 80 → C0.0: (M, AC20, 0080)
C3.0: (I, AC20, 0020)

c. C3: W AC20 ← 80 → C3.0: (M, AC20, 0080)

d. C1: R AC10 → C1.2: (S, AC10, 0010) returns 0010

e. C0: W AC08 ← 48 → C0.1 (M, AC08, 0048)
C3.1: (I, AC08, 0008)

f. C0: W AC30 ← 78 → C0.2: (M, AC30, 0078)
M: AC10 ← 0030 (write-back to memory)

g. C3: W AC30 ← 78 → C3.2 :(M, AC30, 0078)

5.2

5.2 a. C0: R AC20 Read miss, satisfied by memory
C0: R AC28 Read miss, satisfied by C1's cache
C0: R AC20 Read miss, satisfied by memory, write-back 110
Implementation 1: 100 + 40 + 10 + 100 + 10 = 260 stall cycles
Implementation 2: 100 + 130 + 10 + 100 + 10 = 350 stall cycles

b. C0: R AC00 Read miss, satisfied by memory
C0: W AC08 ← 48 Write hit, sends invalidate
C0: W AC20 ← 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. C1: R AC20 Read miss, satisfied by memory
C1: R AC28 Read hit
C1: R AC20 Read miss, satisfied by memory
Implementation 1: 100 + 0 + 100 = 200 stall cycles
Implementation 2: 100 + 0 + 100 = 200 stall cycles

d. C1: R AC00 Read miss, satisfied by memory
C1: W AC08 ← 48 Write miss, satisfied by memory, write back AC28
C1: W AC20 ← 78 Write miss, satisfied by memory
Implementation 1: 100 + 100 + 10 + 100 = 310 stall cycles
Implementation 2: 100 + 100 + 10 + 100 = 310 stall cycles

Case Study 2: Simple Directory-Based Coherence

5.9 (a)

a. i. C3:R, M4
Messages:
- Read miss request message from C3 to Dir4 (011 → 010 → 000 → 100)

- Read response (data) message from M4 to C3 (100→101→111→011)

C3 cache line 0: <I, x, x, …..> → <S, 4, 4, …..>
Dir4: <I, 00000000> →<S, 00001000>, M4 = 4444…..

ii. C3:R, M2
Messages:
- Read miss request message from C3 to Dir2 (011→ 010)
- Read response (data) message from M2 to C3 (010→011)

C3 cache line 0: <S, 4, 4, …..> → <S, 2, 2, …..>
C2 Dir: <I, 00000000> →<S, 00001000>, M4 = 4444…..
Note that Dir4 still assumes C3 is holding M4 because C3 did not notify it that it replaced line 0. C3 informing Dir4 of the replacement can be a useful upgrade to the protocol.

iii. C7: W, M4 ← 0xaaaa
Messages:
- Write miss request message from C7 to M4 (111→ 110→ 100)
- Invalidate message from Dir4 to (100→101→111→011)
- Acknowledge message from C3 to Dir4 (011→ 010→000→100)
- Acknowledge message from Dir4 to C7 (100→ 101→111)

C3 cache line 0: <S, 4, 4, …..> → <I, x, x, …..>
C7 cache line 0: <I, x, x, …..> → <M, aaaa, …..>
Dir4: <S, 00001000> →<M, 10000000>, M4 = 4444…..

iv. C1: W, M4 ← 0xbbbb
Messages:
- Write miss request message from C1 to M4 (001 → 000 → 100)
- Invalidate message from Dir4 to C7 (100 → 101 → 111)
- Acknowledge message (with data write-back) from C7 to Dir4 (111 → 110 →100)
- Write Response (data) message from Dir4 to C1 (100 → 101 →001)

C7 cache line 0: <M, aaaa, …..> → <I, x, x, …..>
C1 cache line 0: <I, x, x, …..> → <M, bbbb, …..>
Dir4: <M, 10000000> →<M, 00000001> M4 =aaaa…..

Example message formats:
No data message: <no data message flag, message type, destination (dir/cache, & number), block/line number>

Data message: <data message flag, message type, destination (dir/cache, & number), block/line number, data>

Case Study III

5.15

Initial values of A, B are 0

P1:	P2:
While (B == 0);	While (A==0);
A = 1;	B = 1;

Under any statement interleaving that does not break the semantics of each thread

when executed individually, the values of A and B will remain 0.

a. Sequential consistency
For example, in the ordering:
P1:While (B == 0);
P1: A = 1;
P2: While (A == 0);
P2: B = 1;
B will remain 0, and neither A nor B will be changed. That will be the case for any SC ordering.

b. Total store order
TSO relaxes W → R ordering in a thread. Neither of these two threads has write followed by a read. So the answer will be like part (a).

5.23

To keep the figures from becoming cluttered, the coherence protocol is split into two parts. Figure S.3 presents the CPU portion of the coherence protocol, and Figure S.4 presents the bus portion of the protocol. In both of these figures, the

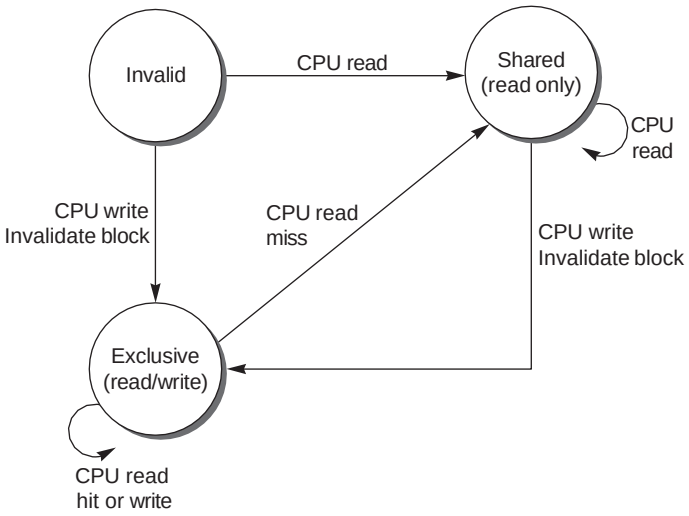

Figure S.3 CPU portion of the simple cache coherency protocol for write-through caches.

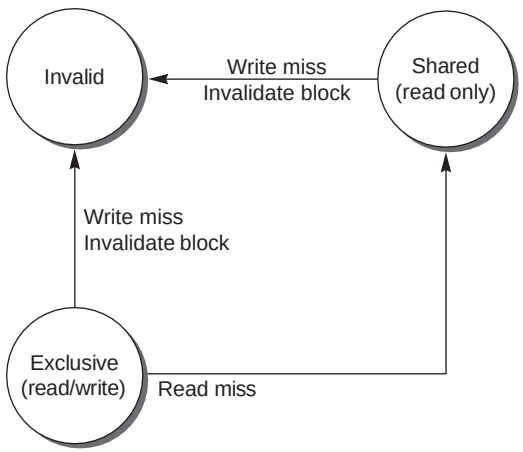

Figure S.4 Bus portion of the simple cache coherency protocol for write-through caches.

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.3 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.4. 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. In addition, 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. It is not possible for valid cache blocks to be incoherent with respect to main memory in a system with write-through caches.

5.25

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 addi- tion, 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 and modified should be written to memory because the contents of invalid words are not neces- sarily coherent with the system.

Finally, given that the state machine of Figure 5.6 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 infor- mation, the only solution is to change the coherence protocol slightly.

5.29

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 direc- tory. 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.