CS252 Graduate Computer Architecture
Multiprocessors and Multithreading
Solutions

November 14, 2007

Problem 1: Directory-based Cache Coherence

Problem 1.A — Cache State Transitions

Complete Table 1.

No.	Current State	Event Received	Next State	Action
1	C-invalid	load	C-transient	load-request -> home
2	C-invalid	store	C-transient	store-request -> home
3	C-invalid	invalidate-request	C-invalid	invalidate-reply -> home
4	C-invalid	shared-copy-request	C-invalid	nothing
5	C-invalid	exclusive-copy-request	C-invalid	nothing
6	C-shared	load	C-shared	processor reads cache
7	C-shared	store	C-transient	store-request -> home
8	C-shared	replace	C-invalid	nothing
9	C-shared	invalidate-request	C-invalid	invalidate-reply -> home
10	C-modified	load	C-modified	processor reads cache
11	C-modified	store	C-modified	processor writes cache
12	C-modified	replace	C-invalid	write-back -> home
13	C-modified	shared-copy-request	C-shared	shared-copy-reply -> home
14	C-modified	exclusive-copy-request	C-invalid	exclusive-copy-reply -> home
15	C-transient	load-reply	C-shared	data -> cache, processor reads cache
16	C-transient	store-reply	C-modified	data -> cache, processor writes cache
17	C-transient	invalidate-request	C-transient	invalidate-reply -> home
18	C-transient	shared-copy-request	C-transient	nothing
19	C-transient	exclusive-copy-request	C-transient	nothing

Table 1: Cache State Transitions

Problem 1.B **Directory State Transitions**

Complete Table 2.

No.	Current State	Message Received	Next State	Action
1	H-uncached	load-request	H-shared[{*k*}]	load-reply -> *k*
2	H-uncached	store-request	H-modified[*k*]	store-reply -> *k*
3	H-shared[*S*]	load-request	H-shared[$S \cup \{k\}$]	load-reply -> *k*
4	H-shared[*S*]	store-request	H-transient[count = |*S*|]	for $i \in S$ invalidate-request -> *i*
5	H-modified[*m*]	load-request	H-transient	shared-copy-request -> *m*
6	H-modified[*m*]	store-request	H-transient	exclusive-copy-request -> *m*
7	H-modified[*m*]	write-back	H-uncached	data -> memory
8	H-transient	load-request	H-transient	retry -> *k*
9	H-transient	store-request	H-transient	retry -> *k*
10	H-transient	write-back	H-uncached	data -> memory, retry -> *j*
11	H-transient[count > 1]	invalidate-reply	H-transient[--count]	nothing
12	H-transient[count = 1]	invalidate-reply	H-modified[*j*]	store-reply -> *j*
13	H-transient	shared-copy-reply	H-shared[{*k*, *j*}]	data -> memory, load-reply -> *j*
14	H-transient	exclusive-copy-reply	H-modified[*j*]	store-reply -> *j*

Table 2: Home Directory State Transitions

Problem 1.C — Protocol Understanding

Consider the situation in which the home site sends an exclusive-copy-request to a site. This can only happen when the home directory shows that the modified copy resides at that site. The home site intends to obtain the most up-to-date data and exclusive ownership, and then supply them to another site that has issued a store-request. In Table 1, the last row (line 19) specifies the PP behavior when the current cache state is C-transient (not C-modified) and an exclusive-copy-request is received.

Give a simple scenario that causes this situation. You should explain your answer clearly.

Assume initially the home directory state is H-modified[m], indicating that the block is modified at site *m*. Consider the following scenario:

1. The home site receives a store-request from a site (not *m*). The home site sends an exclusive-copy-request to site *m*. The store-request is *suspended and buffered* at the home site. The home directory state becomes H-transient.
2. Before the exclusive-copy-request arrives, the modified cache line is replaced at site *m* (due to a cache line conflict). Site *m* sends a write-back to the home site. The cache line state becomes C-invalid. The processor at site *m* issues a load/store instruction accessing the data that has just been replaced, causing site *m* to send a load-request/store-request to the home site. The load/store instruction is suspended at site *m*. The cache line state becomes C-transient.
3. The exclusive-copy-request arrives at site *m*.

Problem 1.D — Non-FIFO Network

FIFO message passing is a necessary assumption for the correctness of the protocol. Assume now that the network is non-FIFO. Give a simple scenario that shows how the protocol fails.

Assume initially that a particular block is not cached by any site. Consider the following scenario:

1. The home site receives a load-request from site A.
2. The home site sends a load-reply to site A and changes the state to H-shared[{A]}.
3. The home site receives a store-request from site B.
4. The home site sends an invalidate-request to site A and changes the state to H-transient.
5. The invalidate-request arrives at site A before the load-reply.
6. Site A sends an invalidate-reply to the home site.
7. The load-reply arrives at site A.
8. The invalidate-reply arrives at the home.
9. The home site sends a store-reply to site B.
10. The store-reply arrives at site B.

Site A has a C-shared copy and site B has a C-modified copy.

Problem 1.E **Replace**

In the current scheme, when a replace operation happens, the PP simply changes the cache state to C-invalid, but does not inform the home node that the local node is no longer a sharer. Explain why this still preserves global cache coherence. Also describe the advantage(s) and disadvantage(s) of this scheme, compared with the scheme of having the PP always inform the home node that the local node is no longer a sharer after every replace operation.

The home does not need to know that a certain node is no longer a sharer to preserve global cache coherence (although it does need to know about additional sharers). This will just cause the home node to issue an unnecessary invalidation-request to that node when another node tries to get exclusive privileges to that memory block, because the home node thinks it is still caching the data, even though it has already evicted the data. The node that is no longer caching the data will just send back an invalidation-reply as usual, without having to invalidate. It will not have a stale copy of the data, because the data is no longer in the cache, and it would have to request the data from the directory anyway whenever it needs to use it again, because it was in the invalid state even before it got invalidated by the home node. The advantage of this scheme is that it reduces network traffic by not having to inform the home node every time something gets evicted from the local cache. The tradeoff is the increased network traffic whenever the home node sends out unnecessary invalidation-requests and the local nodes responds to these requests.

Problem 2: Multithreaded architectures

In this question we will analyze the performance of the following C program on a multi-threaded architecture. You should assume that arrays **A**, **B** and **C** do not overlap in memory.

C code

```
for (i=0; i<328; i++) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

Our machine is a single-issue, in-order processor. It switches to a different thread every cycle using fixed round robin scheduling. Each of the N threads executes one instruction every N cycles. We allocate the code to the threads such that every thread executes every Nth iteration of the original C code.

Integer instructions take 1 cycle to execute, floating point instructions take 4 cycles and memory instructions take 3 cycles. All execution units are fully pipelined. If an instruction cannot issue because its data is not yet available, it inserts a bubble into the pipeline, and retries after N cycles.

Below is our program in assembly code for this machine for a single thread executing the entire loop.

```
loop:  ld f1, 0(r1)      ; f1 = A[i]
       ld f2, 0(r2)      ; f2 = B[i]
       fmul f4, f2, f1   ; f4 = f1 * f2
       st f4, 0(r1)      ; A[i] = f4
       ld f3, 0(r3)      ; f3 = C[i]
       fadd f5, f4, f3   ; f5 = f4 + f3
       st f5, 0(r3)      ; C[i] = f5
       add r1, r1, 4     ; i++
       add r2, r2, 4
       add r3, r3, 4
       add r4, r4, -1
       bnez r4, loop     ; loop
```

Problem 2.A

We allocate the assembly code of the loop to N threads such that every thread executes every Nth iteration of the original loop. Write the assembly code that one of the N threads would execute on this multithreaded machine.

The code for each thread would look very similar to the assembly code above, except the array indices need to be incremented by N instead of 1, and each thread would have a different starting offset when loading/storing from/to arrays A, B, and C.

Problem 2.B

What is the minimum number of threads this machine needs to remain fully utilized issuing an instruction every cycle for our program? Explain.

4, largest latency for any instruction is 4

Problem 2.C

What will be the peak performance in flops/cycle for this program? Explain briefly.

8/48 = 2/12 = 0.17 flops/cycle

Problem 2.D

Could we reach peak performance running this program using fewer threads by rearranging the instructions? Explain briefly.

Yes, we can hide the latency of the floating point instructions by moving the add instructions in between floating point and store instructions – we'd only need 3 threads. Moving the third load up to follow the second load would further reduce thread requirement to only 2.