CS252 Graduate Computer Architecture
Multiprocessors and Multithreading
Practice Problems

November 14, 2007

Problem 1: Directory-based Cache Coherence

Directory-based Cache Coherence Protocol

In this problem we consider a CCDSM (cache-coherent distributed shared memory) system. The system consists of a number of sites connected by an interconnection network. As shown in Figure 1, each *site* has a processor, an L1 cache, a shared memory, and a protocol processing component (PP). The PP implements global cache coherence using a directory-based cache coherence protocol. For each cache line, we maintain a cache state to specify the current coherence state of the cache line. For each memory block, we maintain a directory entry to record the sites that are currently caching that block. For every global address, there is a *home* site where the physical memory and directory entry is maintained. Assume that the home site can be determined by the global address using its most significant bits.

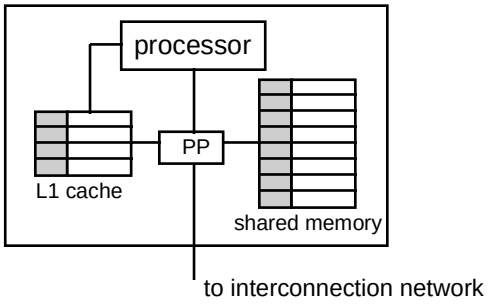

Figure 1: Site Configuration

A simple full-map directory structure is used. Each directory entry keeps a complete record of the sites that are sharing the memory block. The most common implementation keeps a bit-vector in each directory entry. The bit-vector has one bit for each site, indicating if a valid copy of the memory block is cached at that site. A dirty bit is also needed to indicate if the block has been modified. Unlike bus-based snoopy protocols, the directory-based protocol does not rely on broadcast to invalidate stale copies. Instead, because the locations of shared copies are known, cache coherence can be achieved by sending point-to-point protocol messages to only the sites that have cached the accessed memory block. The elimination of broadcast overcomes the major limitation on scaling cache coherent systems to parallel machines with a large number of processors.

The PPs are responsible for servicing memory access instructions, processing protocol messages, and maintaining cache line states and home directory states. When the processor issues a memory access instruction, the PP checks the addressed cache line's state. If the cache state shows that the instruction cannot be completed locally, the PP suspends the instruction, and sends a protocol request message to the corresponding home site. When this request arrives at the

home site, the PP at the home site checks the home directory state, and sends a protocol reply message back to the requesting site to supply the requested data and/or exclusive ownership (in order to do this, the home site may need to obtain, from a remote site, the most up-to-date data and/or exclusive ownership if the memory block has been modified; or invalidate all the shared copies if the memory block is shared and the protocol message received is a store-request). At the requesting site, when the PP receives the protocol reply message, it resumes the suspended memory access instruction.

We make the following assumptions about the interconnection network:

- Message passing is reliable, and free from deadlock, livelock and starvation. In other words, the transfer latency of any protocol message is finite.
- Message passing is FIFO. That is, protocol messages with the same source and destination sites are always received in the same order as that in which they were issued.

Memory instructions: The basic memory access instructions are load and store. A load instruction reads the most up-to-date value of a given location, while a store instruction writes a specific value to a given location. We also need to consider cache replacement operations. A replace operation invalidates a cache line and, if the cache line has been modified, writes the modified data back to memory.

Both load and store are processor-issued instructions. Replace, on the other hand, is normally caused by a load/store instruction when a read/write miss leads to an associative conflict in the cache. In this situation, the load/store instruction cannot be processed before the replace (which is a "side-effect" of the load/store instruction) operation is completed. A cache line in a transient state cannot be replaced.

Cache states: For each cache line, there are 4 possible states:

- C-invalid: The line has no valid data.
- C-shared: The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified: The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
- C-transient: The accessed data is in a *transient* state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home directory states: For each memory block, there are 4 possible states:

- H-uncached: The memory block is not cached by any site. Memory has the most up-to-date data.
- H-shared[*S*]: The memory block is shared by the sites specified in *S* (*S* is a set of sites). The data in memory is also valid.
- H-modified[*m*]: The memory block is exclusively cached at site *m*, and has been modified at that site. Memory does not have the most up-to-date data.
- H-transient: The memory block is in a transient state (for example, the home site has just sent a protocol request to the modified site in order to obtain the most up-to-date data, but has not received the corresponding protocol reply). A counter, *count*, is needed when H-

transient represents a transient state in which the home site is waiting for the acknowledgments to the invalidation requests it has issued.

Protocol messages: There are 12 different protocol messages, which are summarized in the following table (their meaning will become clear later). A protocol message includes the message type, the accessed memory address and, if necessary, the requested or written-back data. A protocol message usually comes in a *request* and *reply* pair. However, there are two exceptions: write-back and retry. Write-back writes a modified cache line back to the main memory. This is a one-way message that does not need a reply (compared with protocol requests, this saves one reply message). Retry is a NAK (negative acknowledgment) message which indicates that something abnormal has happened, and some request cannot be processed and should be retried later. This is possible since the parallel system runs in a distributed way so that operations (e.g. sending a protocol message from one site to another) cannot be treated as atomic operations.

No.	Message Type	Includes data?
1	load-request	no
2	store-request	no
3	shared-copy-request	no
4	exclusive-copy-request	no
5	invalidate-request	no
6	load-reply	yes
7	store-reply	yes
8	shared-copy-reply	yes
9	exclusive-copy-reply	yes
10	invalidate-reply	no
11	write-back	yes
12	retry	no

The behavior of the PP can be defined by two finite state machines: one for cache line states, the other for home directory states. In this problem, we consider a very simple invalidation-based cache coherence protocol that implements the *sequential consistency* memory model. A brief (but neither formal nor complete) description is given below to help you understand the protocol.

Cache state transitions:

When the processor issues a load instruction,

- If the cache state is C-shared or C-modified, the PP supplies the processor the data from the cache. The cache state is not changed.
- If the cache state is C-invalid, the PP suspends the load instruction, and sends a load-request to the accessed memory's home site to request the data. The cache state is changed to C-transient. Later when the corresponding load-reply arrives, the PP places the data in the cache, changes the cache state to C-shared, and resumes the suspended load instruction.

When the processor issues a store instruction,

- If the cache state is C-modified, the PP allows the processor to write to the cache. The cache state is not changed.
- If the cache state is C-invalid or C-shared, the PP suspends the store instruction, and sends a store-request to the accessed memory's home site to request the data and exclusive ownership. The cache state is changed to C-transient. Later when the corresponding store-reply arrives, the PP places the data in the cache, changes the cache state to C-modified, and resumes the suspended store instruction.

When a replace operation happens,

- If the cache state is C-shared, the PP simply changes the cache state to C-invalid.
- If the cache state is C-modified, the PP sends a write-back message to the home site to write the modified data to memory, and changes the cache state to C-invalid.

Home directory state transitions:

When a load-request from site *k* arrives at the home site,

- If the home directory state is H-uncached, the PP sends a load-reply to site *k* to supply the requested data. The directory state is changed to H-shared[*S*], where $S = \{k\}$.
- If the home directory state is H-shared[*S*], the PP sends a load-reply to site *k* to supply the requested data. The directory state is changed to H-shared[*S'*], where $S' = S \cup \{k\}$.
- If the home directory state is H-modified[*m*], the PP sends a shared-copy-request to site *m* in order to obtain the most up-to-date data. The directory state is changed to H-transient. Later when the corresponding shared-copy-reply arrives at the home site, the PP updates memory, sends a load-reply to site *k* to supply the requested data, and then changes the directory state to H-shared[*S*], where $S = \{m, k\}$.

When a store-request from site *k* arrives at the home site,

- If the home directory state is H-uncached, the PP sends a store-reply to site *k* to supply the requested data and exclusive ownership. The directory state is changed to H-modified[*k*].
- If the home directory state is H-shared[*S*], the PP sends an invalidate-request to each of the sites specified in *S*. The directory state is then changed to H-transient, with a dedicated counter initialized to the number of invalidate-requests that have been issued. Later when all the invalidations have been acknowledged, the PP sends a store-reply to site *k*, and changes the directory state to H-modified[*k*].
- If the home directory state is H-modified[*m*], the PP sends a exclusive-copy-request to site *m* in order to obtain the most up-to-date data and exclusive ownership. The directory state is then changed to H-transient. Later when the corresponding exclusive-copy-reply is received, the PP sends a store-reply to site *k*, and changes the directory state to H-modified[*k*].

When a write-back from site *m* arrives at the home site,

- If the home directory state is H-modified[*m*] or H-transient, the PP updates the memory with the write-back data, and changes the directory state to H-uncached.

*Note: The cache state transitions described above are for each physical address at the granularity of the cache line size.

Problem 1.A **Cache State Transitions**

Table 1 shows the cache state transitions of the protocol (note that some tricky transitions are intentionally ignored here). The "current state" is the current cache line state. The "event received" is the event that the PP receives, which can be a load/store instruction, a replace operation, or a protocol message issued from the home site. The "next state" is the next cache line state after the PP processes the received event. The "action" is what the PP must do when processing the received event. This usually includes generating some new protocol message, placing some data in the cache, and so on.

Complete Table 1.

Problem 1.B **Directory State Transitions**

Table 2 shows the home directory state transitions of the protocol (note that some tricky transitions are intentionally ignored here). The "current state" is the current home directory state. The "message received" is the protocol message that the PP receives. The "next state" is the next directory state after the PP processes the received message. The "action" is what the PP must do when processing the received event. This usually includes generating some new protocol message(s), updating memory with the most up-to-date data, and so on. We use *k* to represent the site that issued the received message. For H-transient state, we use *j* to represent the site that issued the *original* protocol request (load-request/store-request).

Complete Table 2.

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.

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.

Problem 1.E

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.

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

Table 1: Cache State Transitions

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]	
3	H-shared[S]	load-request	H-shared[$S \cup \{k\}$]	
4	H-shared[S]	store-request		
5	H-modified[m]	load-request		
6	H-modified[m]	store-request		
7	H-modified[m]	write-back		data -> memory
8	H-transient	load-request		
9	H-transient	store-request		
10	H-transient	write-back		
11	H-transient[count > 1]	invalidate-reply	H-transient[--count]	nothing
12	H-transient[count = 1]	invalidate-reply		
13	H-transient	shared-copy-reply		
14	H-transient	exclusive-copy-reply		

Table 2: Home Directory State Transitions

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.

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.

Problem 2.C

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

Problem 2.D

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