Computer Architecture and Engineering

CS152 Quiz #5

April 29, 2010

Professor Krste Asanovic

Name:____ANSWER KEY____

This is a closed book, closed notes exam.

80 Minutes

10 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with students who have not yet taken the quiz. If you have inadvertently been exposed to the quiz prior to taking it, you must tell the instructor or TA.
- You will receive no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Writing name on each sheet	________	1 Point
Question 1	________	18 Points
Question 2	________	15 Points
Question 3	________	20 Points
Question 4	________	26 Points
TOTAL	________	80 Points

NAME:________________________

Problem Q5.1: Sequential Consistency 18 points

In this problem, we consider the implementation of sequential consistency (SC) in a cache-coherent multiprocessor that uses a snoopy bus. A processor can intervene in a bus transaction by asserting the *retry* signal, causing the processor that initiated the request to attempt it again later. The bus supports only one outstanding cache miss at a time for the entire system.

Each processor is equipped with a non-blocking data cache that supports *hit-under-miss*: while the cache is processing a miss, accesses that hit in the cache can still proceed. The cache does not support *miss-under-miss*, so it will block in the event of a second miss. A processor sends loads and stores to its cache in program order.

Problem Q5.1.A 10 points

Without further modifications, the hit-under-miss scheme can lead to violations of sequential consistency. Carefully explain why. Additionally, provide a short pseudocode sequence for two processors that could result in a sequentially inconsistent execution. Indicate whether each data memory access in your example is a cache hit or miss.

Hit-under-miss enables younger memory accesses that hit in the cache to be made visible before an older memory access that misses in the cache. Another processor could see these accesses out of program order, which violates SC.

Assume that variables X and Y start out as 0. Because the write to Y may proceed before the write to X, the following sequence of accesses could result in r1 = 1, r2 = 0, which is clearly a violation of SC.

Processor 0	Hit/Miss	Processor 1	Hit/Miss
X=1	M	r1=Y	M
Y=1	H	r2=X	H

NAME:_________________________

Problem Q5.1.B **8** points

Fortunately, it is possible to make the hit-under-miss scheme appear to be sequentially consistent by leveraging the snoopy bus. Describe how to implement SC while maintaining hit-under-miss.

Reordering only violates SC if other processors can observe it. The only way for e.g. P1 to observe P0's accesses is via a bus transaction. So, if P1 initiates a bus transaction while P0 is servicing a miss that has been hit-under, P0 can tell P1 to retry until the miss has been resolved. This way, the miss and all subsequent hits are made visible at the same time.

NAME:_________________________

Problem Q5.2: Relaxed Memory Models 15 points

The following code implements a *seqlock*, which is a reader-writer lock that supports a single writer and multiple readers. The writer never has to wait to update the data protected by the lock, but readers may have to wait if the writer is busy. We use a seqlock to protect a variable that holds the current time. The lock is necessary because the variable is 64 bits and thus cannot be read or written atomically on a 32-bit system.

The seqlock is implemented using a sequence number, *seqno*, which is initially zero. The writer begins by incrementing *seqno*. It then writes the new time value, which is split into the 32-bit values *time_lo* and *time_hi*. Finally, it increments *seqno* again. Thus, if and only if *seqno* is odd, the writer is currently updating the counter.

The reader begins by waiting until *seqno* is even. It then reads *time_lo* and *time_hi*. Finally, it reads *seqno* again. If *seqno* didn't change from the first read, then the read was successful; otherwise, the read is retried.

This code is correct on a sequentially consistent system, but on a system with a fully relaxed memory model it may not be. Insert the minimum number of memory fences to make the code correct on a system with a relaxed memory model. To insert a fence, write the needed fence (Membar_{LL}, Membar_{LS}, Membar_{SL}, Membar_{SS}) in between the lines of code below.

Writer

```
LOAD   R_seqno, (seqno)

ADD    R_seqno, R_seqno, 1

STORE  (seqno), R_seqno
Membar_SS

STORE  (time_lo), R_time_lo

STORE  (time_hi), R_time_hi
Membar_SS

ADD    R_seqno, R_seqno, 1

STORE  (seqno), R_seqno
```

Reader

```
Loop: LOAD   R_seqno_before, (seqno)

      IF(R_seqno_before & 1)
           goto Loop
      Membar_LL

      LOAD   R_time_lo, (time_lo)

      LOAD   R_time_hi, (time_hi)
      Membar_LL

      LOAD   R_seqno_after, (seqno)

      IF(R_seqno_before != R_seqno_after)
           goto Loop
```

NAME:________________________

Problem Q5.3: Directory Protocols 20 points

Alice P. Hacker is designing a large-scale cache-coherent multiprocessor. Recalling that directories are more scalable than snoopy buses, Alice decides to implement a full-map directory to maintain coherence.

Problem Q5.3.A 6 points

Alice's first directory design maintains a bit vector for each cache-line-sized block of main memory. For a given memory block, each bit in the bit vector represents whether or not one processor in the machine has that line cached. For a 256-processor system with 64B cache lines and 8GB of main memory, how large would Alice's directory be?

Directory size = (# of cache lines)*(# of processors) bits
= (8GB/64B)*(256) bits
= 32 Gbits
= 4GB

________4GB_______

Problem Q5.3.B 6 points

Alice realizes that her initial design might be impractical to implement. She instead considers a directory organization that supports up to four sharers of a line at a time. If a fifth processor requests a copy of the line, one of the original four sharers is invalidated. For each memory line, this approach requires four processor IDs and four valid bits. Now, how large is Alice's directory?

Directory size = (# of cache lines)*(4(1 + $\log_2$(# of processors))) bits
= (8GB/64B)*(36) bits
= 9*0.5 Gbits
= 576MB

________576MB_______

NAME:_____________________________

Problem Q5.3.C **8 points**

Alice is concerned about the performance of the directory that supports only four sharers at a time. As a final design alternative, she considers also adding a *globally shared* bit to each line. Now, more than four processors can share the line, but if a fifth processor requests a copy of a cache line, the globally shared bit is set. If some processor attempts to write a globally shared line, the directory no longer knows precisely which processors have a copy, so it sends invalidations to all processors in the system.

Consider a program in which all processors in the system are contending for a lock, using a test-and-test-and-set algorithm. The critical section is sufficiently long that all processors not in the critical section are spinning on the lock. Which four-sharers directory organization will perform best: the directory with or without the globally shared bit? Justify your answer.

For your reference, the test-and-test-and-set lock acquisition algorithm works as follows:

```
lock(addr):
        while(test-and-set(addr) != 0)
                while(*addr != 0)
                        ;
```

Consider the globally shared bit scheme first. Assume P0 gets the lock first. Then, P1-P255 will each have one directory transaction to get the line shared, at which point the directory tells P0 to write back. Then, when P0 releases the lock, it upgrades to exclusive and P1-P255 are invalidated. About 256 invalidations/downgrades occur per lock acquisition.

Now, consider the four-sharers scheme. After P0 gets the lock, P1-P255 will try to read it. However, only four of P1-P255 can have the line shared at once, so they will keep invalidating each other, even though no writing is occurring! Thus, the number of invalidations/downgrades per lock acquisition is proportional to the length of the critical section and could be very large.

The global scheme will use substantially less network bandwidth and will thus perform better.

NAME:________________________

Problem Q5.4: Where has all the speedup gone? 26 points

In each of the following problems, we consider a parallel program running on a cache-coherent, shared-memory multiprocessor. Each program is correct as written but suffers poor performance.

Problem Q5.4.A 8 points

The following code performs vector-vector addition. The integer *P* equals the number of processors running the program, and the integer *id* contains a given processor's ID number, which lies in the range [0, *P*-1]. *N* is the vector size; you may assume that *N* is large and that *P* divides *N*.

```
int P, id;
int Z[N], X[N], Y[N];

for(int i = id; i < N; i += P)
    Z[i] = X[i] + Y[i];
```

Unfortunately, false sharing eliminates any hope of parallel speedup. Rewrite the code to avoid false sharing to the extent possible.

False sharing will be essentially eliminated if the processors operate on contiguous chunks (i.e. unit-stride).

```
for(int i = (N/P)*id; i < (N/P)*(id+1); i++)
    Z[i] = X[i] + Y[i];
```

Problem Q5.4.B **8 points**

The following code repeatedly gets data from the *bar* function and pushes it onto a shared stack. When this code is running on only a subset of the processors in a system, all other programs see a noticeable slowdown due to the large number of bus transactions. Rewrite the code to reduce the frequency of bus transactions.

```
shared int stack_lock = initially 0;
shared stack s;

while(1)
{
    int foo = bar();

    while(test_and_set(&stack_lock) != 0)
        ;

    s.push(foo);

    stack_lock = 0;
}
```

Replace the test-and-set lock with a test-and-test-and-set lock. (The code from Q5.3.C could be used.)

NAME:_________________________

Problem Q5.4.C **5 points**

Ben Bitdiddle is computing fast Fourier transforms in parallel, but his parallel speedup is lower than he would like. Careful analysis of the code indicates that true sharing is causing a substantial number of coherence misses.

Which of the following changes to the system will reduce the total number of coherence misses that occur when computing a transform? Circle all that apply, and provide a brief explanation for each item that you circle.

a) Increase cache capacity

b) Reduce the number of processors
This will reduce the number of coherence misses because less sharing will occur. In the limit of 1 processor, no coherence misses can occur.

c) Increase cache associativity

d) Add hardware prefetching
This can hide the effect of coherence misses if data is shared in a pattern that is amenable to prefetching.

Problem Q5.4.D **5 points**

Consider Ben Bitdiddle's parallel transform once again. Which of the following changes to the system will *both* reduce the number of coherence misses that occur *and* improve performance? Circle all that apply, and provide a brief explanation for each.

a) Increase cache capacity

b) Reduce the number of processors
If the system is bus-bandwidth-limited, reducing sharing would reduce total bus traffic, so reducing the number of processors may actually improve performance.

c) Increase cache associativity

d) Add hardware prefetching
If the prefetching is timely and the system isn't bus-bandwidth-limited, timely prefetching will improve performance.

NAME:___________________________

NAME:__________________________

END OF QUIZ

NAME:_________________________

CS152 Computer Architecture and Design

Directory-based Cache Coherence Protocol *4/13/2010*

Before introducing a directory-based cache coherence protocol, 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.

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

- C-invalid (= `Nothing`): The accessed data is not resident in the cache.
- C-shared (= `Sh`): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified (= `Ex`): 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 (= `Pending`): 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:

- R(*dir*): The memory block is shared by the sites specified in *dir* (*dir* is a set of sites). The data in memory is valid in this state. If *dir* is empty (i.e., *dir* = ε), the memory block is not cached by any site.
- W(*id*): The memory block is exclusively cached at site *id*, and has been modified at that site. Memory does not have the most up-to-date data.
- T_R(*dir*): The memory block is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
- T_W(*id*): The memory block is in a transient state waiting for a block exclusively cached at site *id* (i.e., in C-modified state) to make the memory block at the home site up-to-date.

Protocol messages: There are 10 different protocol messages, which are summarized in the following table (their meaning will become clear later).

Category	**Messages**
Cache to Memory Requests	`ShReq, ExReq`
Memory to Cache Requests	`WbReq, InvReq, FlushReq`
Cache to Memory Responses	`WbRep(v), InvRep, FlushRep(v)`
Memory to Cache Responses	`ShRep(v), ExRep(v)`

NAME:___________________________

No.	Current State	Handling Message	Next State	Dequeue Message?	Action
1	C-nothing	Load	C-pending	No	ShReq(id,Home,a)
2	C-nothing	Store	C-pending	No	ExReq(id,Home,a)
3	C-nothing	WbReq(a)	C-nothing	Yes	None
4	C-nothing	FlushReq(a)	C-nothing	Yes	None
5	C-nothing	InvReq(a)	C-nothing	Yes	None
6	C-nothing	ShRep (a)	C-shared	Yes	updates cache with prefetch data
7	C-nothing	ExRep (a)	C-exclusive	Yes	updates cache with data
8	C-shared	Load	C-shared	Yes	Reads cache
9	C-shared	WbReq(a)	C-shared	Yes	None
10	C-shared	FlushReq(a)	C-nothing	Yes	InvRep(id, Home, a)
11	C-shared	InvReq(a)	C-nothing	Yes	InvRep(id, Home, a)
12	C-shared	ExRep(a)	C-exclusive	Yes	None
13	C-shared	(Voluntary Invalidate)	C-nothing	N/A	InvRep(id, Home, a)
14	C-exclusive	Load	C-exclusive	Yes	reads cache
15	C-exclusive	Store	C-exclusive	Yes	writes cache
16	C-exclusive	WbReq(a)	C-shared	Yes	WbRep(id, Home, data(a))
17	C-exclusive	FlushReq(a)	C-nothing	Yes	FlushRep(id, Home, data(a))
18	C-exclusive	(Voluntary Writeback)	C-shared	N/A	WbRep(id, Home, data(a))
19	C-exclusive	(Voluntary Flush)	C-nothing	N/A	FlushRep(id, Home, data(a))
20	C-pending	WbReq(a)	C-pending	Yes	None
21	C-pending	FlushReq(a)	C-pending	Yes	None
22	C-pending	InvReq(a)	C-pending	Yes	None
23	C-pending	ShRep(a)	C-shared	Yes	updates cache with data
24	C-pending	ExRep(a)	C-exclusive	Yes	update cache with data

Table H12-1: Cache State Transitions

NAME:________________________

No.	**Current State**	**Message Received**	**Next State**	**Dequeue Message?**	**Action**
1	R(dir) & (dir = ε)	ShReq(a)	R({id})	Yes	ShRep(Home, id, data(a))
2	R(dir) & (dir = ε)	ExReq(a)	W(id)	Yes	ExRep(Home, id, data(a))
3	R(dir) & (dir = ε)	(Voluntary Prefetch)	R({id})	N/A	ShRep(Home, id, data(a))
4	R(dir) & (id ∉ dir) & (dir ≠ ε)	ShReq(a)	R(dir + {id})	Yes	ShRep(Home, id, data(a))
5	R(dir) & (id ∉ dir) & (dir ≠ ε)	ExReq(a)	Tr(dir)	No	InvReq(Home, dir, a)
6	R(dir) & (id ∉ dir) & (dir ≠ ε)	(Voluntary Prefetch)	R(dir + {id})	N/A	ShRep(Home, id, data(a))
7	R(dir) & (dir = {id})	ShReq(a)	R(dir)	Yes	None
8	R(dir) & (dir = {id})	ExReq(a)	W(id)	Yes	ExRep(Home, id, data(a))
9	R(dir) & (dir = {id})	InvRep(a)	R(ε)	Yes	None
10	R(dir) & (id ∈ dir) & (dir ≠ {id})	ShReq(a)	R(dir)	Yes	None
11	R(dir) & (id ∈ dir) & (dir ≠ {id})	ExReq(a)	Tr(dir-{id})	No	InvReq(Home, dir - {id}, a)
12	R(dir) & (id ∈ dir) & (dir ≠ {id})	InvRep(a)	R(dir - {id})	Yes	None
13	W(id’)	ShReq(a)	Tw(id’)	No	WbReq(Home, id’, a)
14	W(id’)	ExReq(a)	Tw(id’)	No	FlushReq(Home, id’, a)
15	W(id)	ExReq(a)	W(id)	Yes	None
16	W(id)	WbRep(a)	R({id})	Yes	data -> memory
17	W(id)	FlushRep(a)	R(ε)	Yes	data -> memory
18	Tr(dir) & (id ∈ dir)	InvRep(a)	Tr(dir - {id})	Yes	None
19	Tr(dir) & (id ∉ dir)	InvRep(a)	Tr(dir)	Yes	None
20	Tw(id)	WbRep(a)	R({id})	Yes	data-> memory
21	Tw(id)	FlushRep(a)	R(ε)	Yes	data-> memory

Table H12-2: Home Directory State Transitions, Messages sent from site **id**

NAME:________________________

CS152 Computer Architecture and Design

Snoopy Cache Coherence Protocol *4/13/2010*

We introduce an invalidation coherence protocol for write-back caches similar to those employed by the SUN MBus. As in most invalidation protocols, only a single cache may *own* a modified copy of a cache line at any one time. However, in addition to allowing multiple shared copies of clean data, multiple shared copies of modified data may also exist. (Here, modified data refers to data different from memory. When multiple shared copies of modified data exist, one of the caches *owns* the current copy of the data instead of the memory.) All shared copies are invalidated any time a new modified (write) copy is created.

The MBus transactions with which we are concerned are:

- Coherent Read (**CR**): issued by a cache on a read miss to load a cache line.
- Coherent Read and Invalidate (**CRI**): issued by a cache on a write-allocate after a write miss.
- Coherent Invalidate (**CI**): issued by a cache on a write hit to a block that is in one of the shared states.
- Block Write (**WR**): issued by a cache on the write-back of a cache block.
- Coherent Write and Invalidate (**CWI**): issued by an I/O processor (DMA) on a block write (a full block at a time).

In addition to these primary bus transactions, there is:

- Cache to Cache Intervention (**CCI**): used by a cache to satisfy other caches' read transactions when appropriate. A **CCI** intervenes and overrides the answers normally supplied by memory. Data should be supplied using **CCI** whenever possible for faster response relative to the memory. However, only the cache that ***owns*** the data can respond by **CCI.**

The five possible states of a data block are:

- Invalid (**I**): Block is not present in the cache.
- Clean exclusive (**CE**): The cached data is consistent with memory, and no other cache has it.
- Owned exclusive (**OE**): The cached data is different from memory, and no other cache has it. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.
- Clean shared (**CS**): The data has not been modified by the corresponding CPU since cached. Multiple **CS** copies and at most one **OS** copy of the same data could exist.
- Owned shared (**OS**): The data is different from memory. Other **CS** copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the **OE** state.)