CS152/252

Computer Architecture and Engineering

Memory Consistency and Cache Coherence

Solution

Assigned April 5, 2023 Problem Set #5 *Due April 27*

https://inst.eecs.berkeley.edu/~cs152/sp23/

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solution to the problems.

The problem sets also provide essential background material for the exam and the midterms. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets yourself you are unlikely to succeed on the exam or midterms! We will distribute solutions to the problem set on the day after the deadline to give you feedback.

Assignments must be submitted through **Gradescope** by **11:59pm PST** on the specified due date. Late submissions will not be accepted, except for extreme circumstances and with prior arrangement.

Name:

SID:

Problem 1: Sequential Consistency

For this problem we will be using the following sequences of instructions. These are small programs, each executed on a different processor, each with its own cache and register set. In the following **R** is a register and **X** is a memory location. Each instruction has been named (e.g., B3) to make it easy to write answers.

Assume data in location X is initially 0.

Processor A	Processor B	Processor C
A1: ST X, 2	B1: R := LD X	C1: ST X, 7
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

For each of the questions below, please circle the answer and provide a short explanation assuming the program is executing under the SC model. **No points will be given for just circling an answer!**

Problem 1.A

Can X hold value of 8 after all three threads have completed? Please explain briefly.

Yes / No

C1, B1-B6, A1-A4, C2-C4

Problem 1.B

Can X hold value of 9 after all three threads have completed?

Yes / No

All results must be even.

Problem 1.C

Can X hold value of 10 after all three threads have completed?

Yes / No

C1-C4, A1-A4, B1-B6

Problem 1.D

For this particular program, can a processor that reorders instructions but follows local dependencies produce an answer that cannot be produced under the SC model?

Yes / No

All loads/stores must be performed in order within each thread since they involve the same memory address, so no new results are possible.

Problem P5.2: Synchronization Primitives

One of the common instruction sequences used for synchronizing several processors are the LOAD RESERVE/STORE CONDITIONAL pair (from now on referred to as LdR/StC pair). The LdR instruction reads a value from the specified address and sets a local reservation for the address. The StC attempts to write to the specified address provided the local reservation for the address is still held. If the reservation has been cleared the StC fails and informs the CPU.

Problem 2.A

Describe under what events the local reservation for an address *must* be cleared.

Another processor requests write permissions to the same cache line.

Note, however, that this is not necessarily the *only* case in which an implementation might want to clear the reservation. For example, in some popular implementations of LdR/StC like the Arm Cortex-A72 series, *evictions* of the reserved line also cause the reservation to be cleared. This leads to interesting restrictions on what operations can be executed between an LdR and StC while still guaranteeing forward progress (i.e., no operations which may perturb the cache).

Problem 2.B

Is it possible to implement LdR/StC pair in such a way that the memory bus is not affected, i.e., unaware of the addition of these new instructions? Explain.

Yes. WbReq and ShReq are sent normally. The "reservation" is local (probably in the snooper or in the cache itself, although that might take too much resources – very few reservations are needed at the same time for any processor).

Problem 2.C

Give two reasons why the LdR/StC pair of instructions is preferable over atomic read-test-modify instructions such as the TEST&SET instruction.

1. The memory bus does not need to be aware of LdR/StC. The implementation is localized to the processor.
2. The read and write operations are explicitly separated, so LdR potentially does not cause extra bus traffic.
3. There is no need for a separate lock variable.

Problem 3: Relaxed Memory Models

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	**Reader**

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}
 	
ADD Rseqno, Rseqno, 1
 	
Membar_{SS}
 	
STORE (seqno), R_{seqno}
	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

Problem 4: Locking Performance

While analyzing some code, you find that a big performance bottleneck involves many threads trying to acquire a single lock.

Conceptually, the code is as follows:

```
int mutex = 0;

while( true )
{
  noncritical_code( );

  lock( &mutex );
  critical_code( );
  unlock( &mutex );
}
```

Assume for all questions that our processor is using a directory protocol, as described in Handout #6.

Test&Set Implementation

First, we will use the atomic instruction `test_and_set` to implement the *lock(mutex)* and *unlock(mutex)* functions.

In C, the instruction has the following function prototype:

```
int return_value = test_and_set(int* maddr);
```

Recall that `test_and_set` atomically reads the memory address *maddr* and writes a 1 to the location, returning the original value.

Using `test_and_set`, we arrive at the following first-draft implementation for the *lock()* and *unlock()* functions:

```
void inline lock(int* mutex_ptr)
{
   while(test_and_set(mutex_ptr) == 1);
}

void inline unlock(int* mutex_ptr)
{
   *mutex_ptr = 0;
}
```

Problem 4.A — Test&Set, The Initial Acquire

Let us analyze the behavior of `Test&Set` while running 1,000 threads on 1,000 cores.

Consider the following scenario: At the start of the program, the lock is invalid in all caches. Then, every thread executes `Test&Set` once. The first thread wins the lock, while the other threads will find that the lock is taken. How many invalidation messages must be sent when all 1,000 threads execute `Test&Set` once?

1,000 `Test&Sets` are performed in the above scenario.

`Test&Set` is an atomic read-write operation and requires exclusive access to the lock's address. Therefore, each `Test&Set` invalidates the previous core who performed `Test&Set`. However, the first core had no one to invalidate, because the lock was initially uncached. Therefore, 999 invalidation messages were sent.

Invalidations 999

Problem 4.B — Test&Set, Spinning

While the first thread is in the critical section (the "winning thread"), the remaining threads continue to execute `Test&Set`, attempting to acquire the lock. Each waiting thread is able to execute `Test&Set` five times before the winning thread frees the lock. How many invalidation messages must be sent while the winning thread was executing the critical section?

999 cores are spinning, each of which executes `Test&Set` five times for a total of 4995 `Test&Sets`. Each `Test&Set` invalidates the previous core who performed `Test&Set`. Therefore, 4995 invalidation messages are sent.

(This assumes that every thread is interleaved).

Invalidations 4995

Problem 4.C **Test&Set, Freeing the Lock**

How many invalidation messages must be sent when the winning thread frees the lock? Assume the critical section is very long, and all 999 other threads have been waiting to acquire the lock.

Freeing the lock involves writing to the lock's address, which requires invalidating all other cores who have cached that address. Because all of the other cores are spinning on `Test&Set`, and only one core will have the lock address at a time, the winning lock will invalidate only the last core to perform a `Test&Set`.

Invalidations 1

Test&Test&Set Implementation

Since our analysis from the previous parts show that a lot of invalidation messages must be sent while waiting for the lock to be freed, let us instead use a regular load alongside the atomic instruction `test&set` to implement the mutex lock.

```
void inline lock(int* mutex_ptr)
{
   while((*mutex_ptr == 1) || test&set(mutex_ptr) == 1);
}

void inline unlock(int* mutex_ptr)
{
   *mutex_ptr = 0;
}
```

(*Note:* the loop evaluation is short-circuited if the first part is true; thus, `test&set` is only executed if (*mutex_ptr) does not equal 1).

Problem 4.D **Test&Set&Set, The Initial Acquire**

Let us analyze the behavior of `Test&Test&Set` while running 1,000 threads on 1,000 cores.

Consider the following scenario: At the start of the program, the lock is invalid in all caches. Then every thread performs the first `Test` (reading *mutex_ptr*) once. After every thread has performed the first `Test` (which evaluates to *False*, because *mutex* == 0), each thread then executes the atomic `Test&Set` once. Naturally, only one thread wins the lock. How many invalidation messages must be sent in this scenario?

1,000 cores perform the first Test. That requires read permissions and invalidates no cores since the lock is initially invalid. All 1,000 cores end up with a copy of the lock.

Then, all cores execute T&S. The *first* T&S will invalidate the other 999 cores' copy, for 999 invalidations.

The other 999 T&S's will invalidate the previous core to perform T&S, for 999 more invalidations. In total 999+999 invalidations occur.

Invalidations 1998

Problem 4.E **Test&Set&Set, Spinning**

While the first thread is in the critical section, the remaining threads continue to execute `Test&Test&Set`. Each waiting thread is able to execute `Test&Test&Set` five times before the winning thread frees the lock. How many invalidation messages must be sent while the winning thread was executing the critical section?

Once the lock has been acquired by the winning core, the other 999 threads will only see *`mutex == 1`* and not execute the `Test&Set`. Therefore, executing the 4995 `Test&Test&Sets` while waiting for the lock to be freed only requires read permissions.

However, the very *first* `Test&Test&Set` will require downgrading the last core who performed a `Test&Set` operation to the *Shared* state, so it could be argued that 1 invalidation message was sent (technically, a *WbReq* message). So 0 invalidations occurred and 1 downgrade occurred. Either 0 or 1 would be acceptable answers.

Invalidations 0 or 1

Problem 4.F **Test&Set&Set, Freeing the Lock**

How many invalidation messages must be sent when the winning thread frees the lock for the `Test&Test&Set` implementation? Assume the critical section is very long, and all 999 other threads have been waiting to acquire the lock.

Freeing the lock will require invalidating the 999 shared copies held by the spinning threads.

Invalidations 999

Problem 5: Directory-based Cache Coherence Update Protocols

Please refer to Handout #6 (on website) for this problem.
In Handout #6, we examine a cache-coherent distributed shared memory system. Ben wants to convert the directory-based invalidate cache coherence protocol from the handout into an update protocol. He proposes the following scheme.

Caches are write-through, not write allocate. When a processor wants to write to a memory location, it sends a WriteReq to the memory, along with the data word that it wants written. The memory processor updates the memory and sends an UpdateReq with the new data to each of the sites caching the block, unless that site is the processor performing the store, in which case it sends a WriteRep containing the new data.

If the processor performing the store is caching the block being written, it must wait for the reply from the home site to arrive before storing the new value into its cache. If the processor performing the store is not caching the block being written, it can proceed after issuing the WriteReq.

Ben wants his protocol to perform well, and so he also proposes to implement silent drops. When a cache line needs to be evicted, it is silently evicted and the memory processor is not notified of this event.

Note that WriteReq and UpdateReq contain data at the word-granularity, and not at the block-granularity. Also note that in the proposed scheme, memory will always have the most up-to-date data and the state C-exclusive is no longer used.

As in the lecture, the interconnection network guarantees that message-passing is reliable, and free from deadlock, livelock, and starvation. Also as in the lecture, message-passing is FIFO, meaning; each home site keeps a FIFO queue of incoming requests and processes them in the order received.

Problem 5.A — Sequential Consistency

Alyssa claims that Ben's protocol does not preserve sequential consistency because it allows two processors to observe stores in different orders. Describe a scenario in which this problem can occur.

Consider lines X and Y, whose home sites are A and B, respectively.

1. A receives a WriteReq for X, and B receives a WriteReq for Y
2. C and D are both caching X and Y. So both A and B send UpdateReq to C and D.
3. C first receives the UpdateReq for X, then the UpdateReq for Y
4. D first receives first the UpdateReq for Y, then the UpdateReq for X

   C and D can receive the UpdateReq in different orders because they are arriving from different home sites, and FIFO message passing only provides guarantees for messages with the same source and destination.

Noting that many commercial systems do not guarantee sequential consistency, Ben decides to implement his protocol anyway. Fill in the following state transition tables (Table P5.5-1 and Table P5.5-2) for the proposed scheme. (Note: the tables do not contain all the transitions for the protocol).

No.	Current State	Event Received	Next State	Action
1	C-nothing	Load	C-transient	ShReq(id, Home, a)
2	C-nothing	Store	C-transient	WriteReq(id, Home, a)
3	C-nothing	UpdateReq	C-nothing	None
4	C-shared	Load	C-shared	processor reads cache
5	C-shared	Store	C-transient	WriteReq(id, Home, a)
6	C-shared	UpdateReq	C-shared	data → cache
7	C-shared	(Silent drop)	C-nothing	Nothing
8	C-transient	ShRep	C-shared	data → cache, processor reads cache
9	C-transient	WriteRep	C-shared	data → cache, store retires
10	C-transient	UpdateReq	C-transient	data → cache

Table P5.5-1: Cache State Transitions

No.	Current State	Message Received	Next State	Action
1	R(dir) & id ∉ dir	ShReq	R(dir + {id})	ShRep(Home, id, a)
2	R(dir) & id ∉ dir	WriteReq	R(dir + {id})	data → memory, WriteRep(id), UpdateRep(i) for i ≠ id in dir
3	R(dir) & id ∈ dir	ShReq	R(dir)	ShRep(Home, id, a)
4	R(dir) & id ∈ dir	WriteReq	R(dir)	data → memory, WriteRep(id), UpdateRep(i) for i ≠ id in dir

Table P5.5-2: Home Directory State Transitions (N = “is not in”)

Problem 5.C — UpdateReq

After running a system with this protocol for a long time, Ben finds that the network is flooded with UpdateReqs. Alyssa says this is a bug in his protocol. What is the problem and how can you fix it?

Because caches do not notify the home site when a line gets replaced, the set S for a memory block will only increase. Eventually, every site that has ever loaded a particular block will be in the set S for that block, resulting in numerous UpdateReq on the network, even though many of the recipients of the update have already replaced that cache line. The solution is to notify the home site when a cache line is replaced and have the home site remove a site from S when such a notification is received.

Problem 5.D — FIFO Assumption

FIFO message passing is a necessary assumption for the correctness of the protocol. If the network were non-FIFO, it becomes possible for a processor to never see the result of another processor's store. Describe a scenario in which this problem can occur.

Consider a site A:

1. Site A sends ShReq for block X to X's home site.
2. X's home site receives ShReq and issues a ShRep.
3. Site B sends a WriteReq for block X to X's home site.
4. X's home receives WriteReq, and issues an UpdateReq to A
5. The ShReq and UpdateReq are re-ordered in the network
6. The UpdateReq arrives at A. Since A is waiting for ShRep, it is in C-transient. It updates

   its cache with the UpdateReq data but remains in C-transient.

7. The ShRep arrives at A. The cache writes the stale data into its cache and reads the result.

Although A received the UpdateReq, it will never see the result of B's store unless the cache line gets replaced and is re-loaded.

Problem 6: Snoopy Cache Coherent Shared Memory

Please refer to Handout #7 (on website) for this problem.
In this problem, we investigate the operation of the snoopy cache coherence protocol in Handout #7. The following questions are to help you check your understanding of the coherence protocol. You do not need to answer these for credit.

- Explain the differences between **CR, CI,** and **CRI** in terms of their purpose, usage, and the actions that must be taken by memory and by the different caches involved.
- Explain why **WR** is not snooped on the bus.
- Explain the I/O coherence problem that **CWI** helps avoid.

Problem 6.A Where in the Memory System is the Current Value

In Table P5.6-1, P5.6-2, and P5.6-3, column 1 indicates the initial state of a certain address X in a cache. Column 2 indicates whether address X is currently cached in any other cache. (The "cached" information is known to the cache controller only immediately following a bus transaction. Thus, the action taken by the cache controller must be independent of this signal, but state transition could depend on this knowledge.) Column 3 enumerates all the available operations on address X, either issued by the CPU (read, write), snooped on the bus (**CR, CRI, CI**. etc), or initiated by the cache itself (replacement). Some state-operation combinations are impossible; you should mark them as such. (See the first table for examples). In columns 6, 7, and 8 (corresponding to this cache, other caches and memory, respectively), **check all possible locations where up-to-date copies of this data block could exist after the operation in column 3 has taken place** and ignore column 4 and 5 for now. Table P5.6-1 has been completed for you. Make sure the answers in this table make sense to you.

Problem 6.B MBus Cache Block State Transition Table

In this problem, **we ask you to fill out the state transitions in Column 4 and 5.** In column 5, fill in the resulting state after the operation in column 3 has taken place. In column 4, list the necessary MBus transactions that are issued by the cache as part of the transition. Remember, the protocol should be optimized such that data is supplied using **CCI** *whenever possible*, and only the cache that ***owns*** a line should issue **CCI**.

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
Invalid	no	none	none	**I**			yes
		CPU read	**CR**	**CE**	yes		yes
		CPU write	**CRI**	**OE**	yes		
		replace	none	*impossible*			
		CR	none	**I**		yes	yes
		CRI	none	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes
Invalid	yes	none	same as above	**I**		yes	yes
		CPU read		**CS**	yes	yes	yes
		CPU write		**OE**	yes		
		replace		*impossible*			
		CR		**I**		yes	yes
		CRI		**I**		yes	
		CI		**I**		yes	
		WR		**I**		yes	yes
		CWI		**I**			yes

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**	yes		yes
		CPU read	none	**CE**	yes		yes
		CPU write	none	**OE**	yes		
		replace	none	**I**			yes
		CR	none or CCI[1]	**CS**	yes	yes	yes
		CRI	none or CCI[1]	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes

Table P5.7-1

[1] Some Sun MBus implementations perform CCI from the cleanExclusive state, while others do not. We accept both answers.

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
ownedExclusive	no	none	none	**OE**	yes		
		CPU read	none	**OE**	yes		
		CPU write	none	**OE**	yes		
		replace	WR	**I**			yes
		CR	CCI	**OS**	yes	yes	
		CRI	CCI	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanShared	no	none	none	**CS**	yes		yes
		CPU read	none	**CS**	yes		yes
		CPU write	CI	**OE**	yes		
		replace	none	**I**			yes
		CR	none[2]	**CS**	yes	yes	yes
		CRI	none	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes
cleanShared	yes	none	same as above	**CS**	yes	yes	yes
		CPU read		**CS**	yes	yes	yes
		CPU write		**OE**	yes		
		replace		**I**		yes	yes
		CR		**CS**	yes	yes	yes
		CRI		**I**		yes	
		CI		**I**		yes	
		WR		**CS**	yes	yes	yes
		CWI		**I**			yes

Table P5.7-2

[2] Some Sun MBus implementations perform CCI from the cleanShared state. However, in these implementations, requests are not broadcast on a bus, but are handled by a central system controller. The system controller arbitrates which cache with a cleanShared copy provides the data. Unless an explanation is provided, CCI is not a valid response from this state.

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedShared	no	none	none	**OS**	yes		
		CPU read	none	**OS**	yes		
		CPU write	CI	**OE**	yes		
		replace	WR	**I**			yes
		CR	CCI	**OS**	yes	yes	
		CRI	CCI	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes
ownedShared	yes	none		**OS**	yes	yes	
		CPU read		**OS**	yes	yes	
		CPU write		**OE**	yes		
		replace	same as above	**I**		yes	yes
		CR		**OS**	yes	yes	
		CRI		**I**		yes	
		CI		**I**		yes	
		WR		*impossible*			
		CWI		**I**			yes

Table P5.7-3