CS152
Computer Architecture and Engineering

Memory Consistency and Cache Coherence
Problem Set #5

Assigned April 13 *Due April 22*

http://inst.eecs.berkeley.edu/~cs152/sp10

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 solutions to the problems.

The problem sets also provide essential background material for the quizzes. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets you are unlikely to succeed at the quizzes! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Homework will not be accepted once solutions are handed out.

Problem P5.1: 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, 1	B1: R := LD X	C1: ST X, 6
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 P5.1.A

Can X hold value of 4 after all three threads have completed? Please explain briefly.

Yes / No

Problem P5.1.B

Can X hold value of 5 after all three threads have completed?

Yes / No

Problem P5.1.C

Can X hold value of 6 after all three threads have completed?

Yes / No

Problem P5.1.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

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 P5.2.A

Describe under what events the local reservation for an address is cleared.

Problem P5.2.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

Problem P5.2.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.

Problem P5.2.D

LdR/StC pair of instructions were conceived in the context of snoopy busses. Do these instructions make sense in our directory-based system in Handout #6? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

Problem P5.3: Directory-based Cache Coherence Invalidate Protocols

In this problem we consider a cache-coherence protocol presented in Handout #6.

Problem P5.3.A — Protocol Understanding

Consider the situation in which memory sends a FlushReq message to a processor. This can only happen when the memory directory shows that the exclusive copy resides at that site. The memory processor intends to obtain the most up-to-date data and exclusive ownership, and then supply it to another site that has issued a ExReq. Table H12-1 row 21 specifies the PP behavior when the current cache state is C-pending (not C-exclusive) and a FlushReq is received.

Give a simple scenario that causes this situation.

Problem P5.3.B — 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 P5.3.C — Replace

In the current scheme, when a cache wants to voluntarily invalidate a shared cache line, the PP informs the memory of this operation. Describe a simple scenario where there would be an error, if the line was "silently dropped." Can you provide a simple fix for this problem in the protocol? Give such a fix if there is one, or explain why it wouldn't be a simple fix.

Problem P5.4: Directory-based Cache Coherence Update Protocols

In Handout #6, we examined 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.

Each home site keeps a FIFO queue of incoming requests, and processes these in the order received.

Problem P5.4.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.

Problem P5.4.B **State Transitions**

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 P5.4-1 and Table P5.4-2) 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		
3	C-nothing	UpdateReq		
4	C-shared	Load	C-shared	processor reads cache
5	C-shared	Store		
6	C-shared	UpdateReq		
7	C-shared	(Silent drop)		Nothing
8	C-transient	ShRep		data → cache, processor reads cache
9	C-transient	WriteRep		
10	C-transient	UpdateReq		

Table P5.4-1: Cache State Transitions

No.	Current State	Message Received	Next State	Action
1	R(dir) & id $\notin$ dir	ShReq	R(dir + {id})	ShRep(Home, id, a)
2	R(dir) & id $\notin$ dir	WriteReq		
3	R(dir) & id $\in$ dir	ShReq		ShRep(Home, id, a)
4	R(dir) & id $\in$ dir	WriteReq		

Table P5.4-2: Home Directory State Transitions

Problem P5.4.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?

Problem P5.4.D **FIFO Assumption**

As in P5.3, 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.

Problem P5.5: Snoopy Cache Coherent Shared Memory

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.

- 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 P5.5.A Where in the Memory System is the Current Value

In Table P5.5-1, P5.5-2, and P5.5-3, 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 P5.5-1 has been completed for you. Make sure the answers in this table make sense to you.

Problem P5.5.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**.

Problem P5.5.C Adding atomic memory operations to MBus

We have discussed the importance of atomic memory operations for processor synchronization. In this problem you will be looking at adding support for an atomic fetch-and-increment to the MBus protocol.

Imagine a dual processor machine with CPUs A and B. Explain the difficulty of CPU A performing fetch-and-increment(x) when the most recent copy of x is cleanExclusive in CPU B's cache. You may wish to illustrate the problem with a short sequence of events at processor A and B.

Fill in the rest of the table below as before, indicating state, next state, where the block in question may reside, and the CPU A and MBus transactions that would need to occur atomically to implement a fetch-and-increment on processor A.

State	other cached	ops	actions by this cache	next state	this cache	other caches	mem
Invalid	yes	read					
		write					

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

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**			
		CPU read					
		CPU write					
		replace					
		CR		**CS**			
		CRI					
		CI					
		WR					
		CWI					

Table P5.5-1

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

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

Table P5.5-2

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

Table P5.5-3