CS252 Graduate Computer Architecture
Multiprocessor Practice Solutions

November 6, 2007

Snoopy Cache Coherence Protocol

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. When multiple shared copies of modified data exist, one of the caches *owns* the cache line instead of the memory, and the other caches have a copy of the owning cache's data. All shared copies are invalidated any time a new modified (write) copy is created.

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.)

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.**

Problem 1: Snoopy Cache Coherent Shared Memory

This problem refers to the snoopy cache coherence protocol described above.

Problem 1.A — Where in the Memory System is the Current Value

In Tables 1.A-1 to 1.A-5, on the following pages, 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). When a cache initiates a replacement, it first writes back dirty data (if any) and then invalidates the block. 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.** (We do not know if the data block *definitely* exists at some locations because we don't know the cache states of the other caches.) The first table has been completed for you. Make sure the answers in this table make sense to you.

Problem 1.B — MBus Cache Block State Transition Table

In this problem, **we ask you to fill out the state transitions in Column 4 and 5 of Tables 1.A-2 to 1.A-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 (in OE or OS) should issue **CCI**.

Table 1.A-1

initial state	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**			√

Table 1.A-2

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

Table 1.A-3

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

Table 1.A-4

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

Table 1.A-5

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

Problem 1.C — Adding atomic memory operations to MBus

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

a) 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.

The problem is that CPU A cannot prevent CPU B from writing to location x between the time CPU A reads location x and the time CPU A writes location x. For fetch and increment to be atomic, CPU A must be able to read and write x without any other processor writing x in between.

For example, consider the following sequence of events and corresponding state transitions and operations:

Event	CPU A	CPU B
1	Read(x); I->CS; send CR	
2		Snoop CR; CE->CS
3		Write(x); CS->OE; send CI
4	Snoop CI; CS->I	
5	Write(x); I->OE; send CRI	
6		Snoop CRI; OE->I

Note that it is OK for CPU B to read x in between the time CPU A reads x and then writes x because this case is the same as if CPU B read x before CPU A did the fetch and increment. Therefore the fetch and increment still appears atomic.

b) What set of cache state transitions and MBus transactions need to occur atomically in order to implement the fetch-and-increment on processor A? 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.

state	cached	ops	actions by this cache	next state	this cache	other caches	mem
Invalid	yes	read	**CR**	**CS**	√	√	√
cleanShared	Yes	write	**CI**	**OE**	√		

c) Operations can occur atomically if the cache controller can lock the bus to prevent other caches from initiating transactions. From which initial cache block states is locking the bus unnecessary?

Assume that CPU A wants to perform an atomic fetch-and-increment on location x. In order to avoid locking the bus, it must be true that no other cache can write to the cache block without the 'approval' of CPU A's cache. When CPU A's cache has the block ownedExclusive, no other cache or memory has the most recent copy of the data. Therefore if any other cache wants to write to x, it would need to receive a CCI from CPU A's cache. If CPU A's cache is in the middle of an atomic operation when it receives a write request (CRI) from another cache, it can just delay sending the CCI until CPU A has completed the atomic operation. Therefore it is not necessary to lock the bus when the initial cache block state is ownedExclusive. If CPU A's cache were in any other initial cache state (I, CS, CE, OS), another cache that wants to write the block either already has the block cached or it can obtain the data from memory or another cache. In these cases, there would be no way CPU A's cache can prevent the other cache from issuing and completing a transaction to write the block. Therefore we can only avoid locking the bus when the initial cache state is ownedExclusive.

Note this example assumes CPU A's cache only delays the CCI response for a short time while the atomic instruction completes. In general, there will be an upper limit on the time by which the cache must respond to ensure the CCI happens correctly. Alternatively, the bus protocol might allow CPU A to say "not ready, please retry", which would force the requestor to repeat their request. Note this is still better than locking the bus because the retry only happens if the snoop matches, not for every access.