

ECE/CS 757 ADVANCED COMPUTER ARCHITECTURE II
Homework 2; Spring 2003
3/3/03

Acknowledgement: Thank you to the student that allowed me to use their assignment as the starting point for these solutions.

1. Text Problem 5.3

For each of the two cases, we calculate the penalties due to the memory system stalls. We can then calculate the percentage of bandwidth used by the processor (because the memory system stalls and the bus cycles are equivalent). From this, we can estimate the number of processors the bus can support (ignoring queuing effects at the bus due to contention).

Because the bus cycle time is twice the processor cycle time, it is easiest to use either bus cycles OR processor cycles for all of the calculations. Here we use bus cycles and adopt the convention of using “~” in place of “cycle.”

Write-through:

It is important to determine the number of cycles for each type of hit and miss.

Instruction and data read misses take 5 cycles. It takes 1 cycle to present the address to the memory, 2 to wait for the memory system to begin responding and then another 2 to transfer a cache line. (16 bytes/line / (64bits/cycle x 8 bytes/bit)).

A write hit takes one cycle to present both address and data to the memory system.

A write miss takes 6 cycles (5 to read the newly allocated line and 1 to write the new data).

These are summarized as follows:

Bus cycles

read: 1~ address + 2~ memory latency + 2~ data transfer = 5~

write miss: 1~ address + 2~ mem latency + 2~ read + 1~ write = 6~

write hit: 1~ address overlapped with 1~ data transfer = 1~

(per 100 instructions)

i-fetch: $100 * 0.015 * 5 = 7.50$

pvt read: $70 * 0.5 * 0.03 * 5 = 5.25$

pvt write: $20 * 0.5 * 0.03 * 6 = 1.80$

$20 * 0.5 * 0.97 * 1 = 9.70$

shd read: $8 * 0.5 * 0.05 * 5 = 1.00$

shd write: $2 * 0.5 * 0.05 * 6 = 0.30$

$2 * 0.5 * 0.95 * 1 = 0.95$

total: 26.50~

For 100 original CPU instructions, we have

2 processor cycles/instruction x 100 instructions x 1 bus cycle/processor cycle = 100 bus cycles

Including memory stall cycles, this is 126.5 bus cycles.

So a single processor uses $26.5/(100+26.5) = 20.9\%$ of the bus bandwidth.

If we ignore queuing effects due to bus contention at high occupancy, we can support $(100\%/20.9\%) = 4.77 \Rightarrow 4$ processors

An easier way to get the same result:

$$126.5 / 26.5 = 4.77 \Rightarrow 4 \text{ processors}$$

Write-back: (assuming write allocate)

Now the write hits take no bus cycles. Each cache miss though has the possibility of a writeback that takes 2 cycles (the whole cache line must be written back.)

Bus cycles (also miss penalties):

read: 1~ address + 2~ memory latency + 2~ data transfer = 5~

write: 1~ address + 2~ memory latency + 2~ read = 5~

writeback: 1~ address overlapped with 2~ data transfer = 2~

(per 100 instructions)

i-fetch: $100 * 0.015 * 5 \sim = 7.50 \sim$

pvt read: $70 * 0.5 * 0.03 * 5 \sim = 5.25 \sim$

pr wrtbak: $70 * 0.5 * 0.009 * 2 \sim = 0.63 \sim$

pvt write: $20 * 0.5 * 0.03 * 5 \sim = 1.50 \sim$

pw wrtbak: $20 * 0.5 * 0.009 * 2 \sim = 0.18 \sim$

shd read: $8 * 0.5 * 0.05 * 5 \sim = 1.00 \sim$

sr wrtbak: $8 * 0.5 * 0.015 * 2 \sim = 0.12 \sim$

shd write: $2 * 0.5 * 0.05 * 5 \sim = 0.25 \sim$

sw wrtbak: $2 * 0.5 * 0.015 * 2 \sim = 0.03 \sim$

total: 16.46~

So for 100 original CPU instructions, we have 116.5 bus cycles.

Thus we can support: $116.5/16.5 = 7.07 \Rightarrow 7$ processors

2. Text Problem 5.4

Stream 1						MESI: Total 397 Cycles					Dragon: Total 515 Cycles					
Action	P1	P2	P3	Bus	Cost	P1	P2	P3	Bus	Cost	P1	P2	P3	Bus	Cost	
r1	E	-	-	BusRd	90	E	-	-	BusRd	90	E	-	-	BusRd	90	
w1	M	-	-	-	1	M	-	-	-	1	M	-	-	-	1	
r1	M	-	-	-	1	M	-	-	-	1	M	-	-	-	1	
w1	M	-	-	-	1	M	-	-	-	1	M	-	-	-	1	
r2	S	S	-	BusRd	90	Sm	Sc	-	BusRd	90	Sm	Sc	-	BusRd	90	
w2	I	M	-	BusUpd	60	Sc	Sm	-	BusUpd	60	Sc	Sm	-	BusUpd	60	
r2	I	M	-	-	1	Sc	Sm	-	-	1	Sc	Sm	-	-	1	
w2	I	M	-	-	1	Sc	Sm	-	BusUpd	60	Sc	Sm	-	BusUpd	60	
r3	I	S	S	BusRd	90	Sc	Sm	Sc	BusRd	90	Sc	Sm	Sc	BusRd	90	
w3	I	I	M	BusUpd	60	Sc	Sc	Sm	BusUpd	60	Sc	Sc	Sm	BusUpd	60	
r3	I	I	M	-	1	Sc	Sc	Sm	-	1	Sc	Sc	Sm	-	1	
w3	I	I	M	-	1	Sc	Sc	Sm	BusUpd	60	Sc	Sc	Sm	BusUpd	60	

Stream 2						MESI: Total 841 Cycles					Dragon: Total 573 Cycles				
Action	P1	P2	P3	Bus	Cost	P1	P2	P3	Bus	Cost	P1	P2	P3	Bus	Cost
r1	E	-	-	BusRd	90	E	-	-	BusRd	90	E	-	-	BusRd	90
r2	S	S	-	BusRd	90	Sc	Sc	-	BusRd	90	Sc	Sc	-	BusRd	90
r3	S	S	S	BusRd	90	Sc	Sc	Sc	BusRd	90	Sc	Sc	Sc	BusRd	90
w1	M	I	I	BusUpgd	60	Sm	Sc	Sc	BusUpd	60	Sm	Sc	Sc	BusUpd	60
w2	I	M	I	BusRdX	90	Sc	Sm	Sc	BusUpd	60	Sc	Sm	Sc	BusUpd	60
w3	I	I	M	BusRdX	90	Sc	Sc	Sm	BusUpd	60	Sc	Sc	Sm	BusUpd	60
r1	S	I	S	BusRd	90	Sc	Sc	Sm	-	1	Sc	Sc	Sm	-	1
r2	S	S	S	BusRd	90	Sc	Sc	Sm	-	1	Sc	Sc	Sm	-	1
r3	S	S	S	-	1	Sc	Sc	Sm	-	1	Sc	Sc	Sm	-	1
w3	I	I	M	BusUpgd	60	Sc	Sc	Sm	BusUpd	60	Sc	Sc	Sm	BusUpd	60
w1	M	I	I	BusRdX	90	Sm	Sc	Sc	BusUpd	60	Sm	Sc	Sc	BusUpd	60

Stream 3						MESI: Total 514 Cycles					Dragon: Total 631 Cycles				
Action	P1	P2	P3	Bus	Cost	P1	P2	P3	Bus	Cost	P1	P2	P3	Bus	Cost
r1	E	-	-	BusRd	90	E	-	-	BusRd	90	E	-	-	BusRd	90
r2	S	S	-	BusRd	90	Sc	Sc	-	BusRd	90	Sc	Sc	-	BusRd	90
r3	S	S	S	BusRd	90	Sc	Sc	Sc	BusRd	90	Sc	Sc	Sc	BusRd	90
r3	S	S	S	-	1	Sc	Sc	Sc	-	1	Sc	Sc	Sc	-	1
w1	M	I	I	BusUpgd	60	Sm	Sc	Sc	BusUpd	60	Sm	Sc	Sc	BusUpd	60
w1	M	I	I	-	1	Sm	Sc	Sc	BusUpd	60	Sm	Sc	Sc	BusUpd	60
w1	M	I	I	-	1	Sm	Sc	Sc	BusUpd	60	Sm	Sc	Sc	BusUpd	60
w1	M	I	I	-	1	Sm	Sc	Sc	BusUpd	60	Sm	Sc	Sc	BusUpd	60
w2	I	M	I	BusRdX	90	Sc	Sm	Sc	BusUpd	60	Sc	Sm	Sc	BusUpd	60
w3	I	I	M	BusRdX	90	Sc	Sc	Sm	BusUpd	60	Sc	Sc	Sm	BusUpd	60

The Illinois MESI protocol does perform very well when there isn't fine grain sharing (i.e. when there is only a single reader/writer at a time), as in Stream 1. That is, when a processor gets the data, it can do its thing, and then pass the data on to the next processor. In this scenario, the Dragon protocol spends a lot of time performing unnecessary updates. However when there is finer grain sharing such as Stream 2, a MESI protocol spends a lot of time thrashing, while the Dragon protocol performs *useful* updates more cheaply. In a situation such as Stream 3, where the processors go through phases together (all reading, then all writing, etc.), the protocols perform similarly. Some unnecessary updates may occur in the Dragon protocol when one processor writes more than once in a row, but a similar negative effect can occur from flushes in the MESI protocol when processors are all trying to write.

3. One of the sufficient conditions for SC given in lecture is: "After load, issuing proc waits for load to complete and store that produced value to complete, before issuing next op" Give an example that shows why the second part ("store that produced value to complete") is included in the condition.

Assume proc A is issuing the load, (LD1) and proc B is issuing the store (ST1). If proc A is allowed to issue store ST2 (to say, a different location) before ST1 is performed with respect to proc C, C could potentially load the new value from the location written by ST2 while seeing the

old value of the location written by ST1. That would cause the following code to print 0 rather than 1, as required by Lamport's definition of SC:

(a,b initialized to 0)

Proc A	Proc B	Proc C
LD1: while(a==0)	ST1: a=1	while(b==0) sees ST2
ST2: b=1		print a but not ST1

If the sufficient condition is enforced, ST2 cannot occur before Proc C can see ST1 is complete.

4. Text Problem 5.12a,b

The sequence of events and resulting numbers of bus transactions varies based on whether or not you have a specific test&set instruction in hardware or if you implement test&set using a Load Linked-Store Conditional pair.

In both cases, I assume that transactions will gain the bus in a first come first served fashion.

Hardware Test&Set

Part A:

In this case there will be many more transactions than in the LL-SC case. The initial steps are:

1. All 16 processors will issue a BusRead to test the lock
2. All 16 processors will see that the lock is cleared and issue a BusReadX in an attempt to test&set. Only 1 processor will successfully read the lock and set it.
3. *At this point, there is a race condition. Either the successful processor will clear the lock or one of the other processors' outstanding BusReadX transactions will invalidate the block before the winner has had a chance to clear it. We will assume that one of the other processors' BusReadX transactions will invalidate the block.*
4. Therefore, the holder will need to issue a BusReadX to regain an exclusive copy of the lock so that it can release the lock.
5. *Let us also assume that the processor that invalidated the holder's copy hasn't had time to see that the lock is held. Thus there are currently 15 BusReadX's waiting to acquire the bus, 14 for test&sets and 1 to clear the lock.*
6. Now, each of the remaining 15 test&sets will fail because they will see the lock as held. They will all then issue a BusRead to begin looping on the shared copy test. The holder will gain exclusive access to the lock and release it before these 15 BusReads gain the bus.
7. Steps 1-6 will repeat for 15 → 1 processors remaining.

Therefore:

$$N = 16 + \sum_{i=1}^{16} 2i = 288 \text{ bus transactions}$$

However, if we don't assume #5, then the worst case can be:

$$N = 16 + \sum_{i=1}^{16} i^2 = 1512 \text{ bus transactions}$$

Yet, if we switch #3 to assuming that the winner of the lock can release it before its copy is invalidated, we will see a best case of:

$$N = 16 + 16 = 32 \text{ bus transactions}$$

Key point to note:

- Once either the BusRead or BusReadX is issued by the processor, it must eventually gain the bus and return the line to that processors cache.

Part B:

In the case where assumptions in steps 3 & 5 are taken, it will take 1650 cycles for the first processor to acquire and release the lock. It will take 14440 cycles for the last processor to release the lock.

Load Locked – Store Conditional**Part A:**

The semantics of a load locked-store conditional pair changes how the initial steps will progress.

1. First all 16 processors will issue BusRead transactions to perform the initial test as in step 1 above.
2. All 16 will see that the lock is available and begin the test&set with a load locked. Since all of them already have a shared copy of the lock, no bus transaction is required at this step. *Assuming that we don't use the "optimized" version of load locked mentioned on top of p.393 of C&S.*
3. By definition of store conditional (as clarified in second to last paragraph of p392 in C&S) only one store conditional will succeed and only one invalidate will be sent out. *Since the critical section is empty, let us assume that the winner of the lock also releases the lock before any of the other processors request a new shared copy.*
4. As the other 15 processors are still spinning, they will all issue BusRead transactions (beginning again with step 1) once they see the invalidation caused by the single successful store conditional. 1-3 will repeat until only 1 processor remains, gets the lock and releases it.

Therefore we will see:

$$N = 16 + \sum_{i=1}^{16} i = 152 \text{ bus transactions}$$

Part B:

It will take 850 cycles for the first processor to acquire and release the lock, 800 for the 16 tests and 50 for the successful test&set.

It will take 7600 cycles before the last processor releases the lock

5. Text problem 6.5

a) If the processor accesses blocks that map to the same set in both the L1 and L2 in this order:

1, 2, 1, 3, 1, 4, 1, 5

the L1 miss stream will be 1, 2, 3, 4, 5 contents of the L1 set will be 1, 5, but the L2 set will contain 2,3,4,5.

b) The same stream results in the same problem, except that the L2 set now only contains 4,5.

6. Text problem 6.6

a) Inclusion is not maintained with the split caches automatically: assume that the Icache misses on a block, and then the Dcache frequently misses on blocks that all map to the same set in the L2. If the L2 uses LRU, it will replace the entire set with the recent misses from the Dcache, and replace the block which resides in the Icache.

b) Again, inclusion is not maintained automatically, now because of the different block sizes. Fundamentally, this problem is similar to that above. In this case, the even words map to group of locations in the L1, and the odd words to a separate group -- though they are the same in the L2. Thus, if the L1 misses on say word 0, and then misses on a whole bunch of odd words (not including word 1) that all map to the same set in the L2, the L2 will not contain word 0, but the L1 still will.

7. Text problem 6.19

The following assumes a write-back L1 (and L2) cache, that inclusion is enforced, and that both the L3 (connected to the bus) and the L2 maintain inclusion bits for the higher level cache in order to filter out coherence traffic.

The L2 still sees all relevant bus traffic forwarded by the L3. The state diagram needs to be modified since the L2 does not directly see processor reads and writes, only read, read-exclusive, and flushes from the L1. These will be denoted *L1Rd*, *L1RdX*, *L1Flush*, respectively. A *modified-but-stale (MI)* state is now needed for the middle cache (and the L3 as well) to indicate that a block has been modified by the processor, but its own copy is stale (invalid). This state is entered when the L1 performs a ReadX. The block enters the M state (in the L2) when the L1 flushes (i.e. writes-back) the block for reasons not caused by a bus request.

The transitions are also modified to be *L1-or-L3-action/L1-effect/L3-effect*, where *L1-or-bus-action* is the action that caused the transition, either from the L1 or from the L3; *L1-effect* is the response sent to the L1 as a result of the action; and *L3-effect* is the action forwarded on to the L3. Note the listed effects include only additional coherence traffic: necessary data transfer is assumed to occur. Prime (') effects are only required if the inclusion bit is set for that block.

Actions from the L3 include *Rd* and *RdX* as usual, where the L2 needs to respond with data (or forward the request on to the L1), but now also include *Inv* (invalidate), and *Share*, which are sent when the L3 is able to supply the data if necessary, but still needs to inform the higher level caches of the coherence.

Misses in the cache can be assumed to start in the *I* (invalid) state.

