CS315a: Parallel Programming and Architecture
Problem Set 2 Solutions

1) Write-through and write-back caches:

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 queueing 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."

a) 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:

Event	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~

Giving the following number of bus cycles per 100 instructions:

Event	Bus Cycles (per 100 instructions)
i-fetch	100 * 0.015 * 5~ = 7.50~
private read	70 * 0.5 * 0.03 * 5~ = 5.25~
private write + allocation	20 * 0.5 * 0.03 * 6~ = 1.80~
private write-through	20 * 0.5 * 0.97 * 1~ = 9.70~
shared read	8 * 0.5 * 0.05 * 5~ = 1.00~
shared write + allocation	2 * 0.5 * 0.05 * 6~ = 0.30~
shared write-through	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 1/2 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 => 4 processors

An easier way to get the same result:
126.5 / 26.5 = 4.77 => 4 processors

b) 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.)

Event	Bus cycles (also miss penalties)
read	1~ address + 2~ memory latency + 2~ data transfer = 5~
write	1~ address + 2~ mem latency + 2~ read = 5~
writeback	1~ address overlapped with 2~ data transfer = 2~

Giving the following number of bus cycles per 100 instructions:

Event	Bus cycles (per 100 instructions)
Instruction fetch	100 * 0.015 * 5~ = 7.50~
private read	70 * 0.5 * 0.03 * 5~ = 5.25~
pr writeback	70 * 0.5 * 0.009 * 2~ = 0.63~
private write	20 * 0.5 * 0.03 * 5~ = 1.50~
pw writeback	20 * 0.5 * 0.009 * 2~ = 0.18~
shared read	8 * 0.5 * 0.05 * 5~ = 1.00~
sr writeback	8 * 0.5 * 0.015 * 2~ = 0.12~
shared write	2 * 0.5 * 0.05 * 5~ = 0.25~
sw writeback	2 * 0.5 * 0.015 * 2~ = 0.03~
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 => 7 processors

2) MESI vs. Dragon:

Stream 1

MESI

	cycles	proc 1	proc 2	proc 3
r1	90	E	-	-
w1	1	M	-	-
r1	1	M	-	-
w1	1	M	-	-
r2	90	S	S	-
w2	1	I	M	-
r2	1	I	M	-
w2	1	I	M	-
r3	90	I	S	S
w3	1	I	I	M
r3	1	I	I	M
w3	1	I	I	M
	279			

Dragon

cycles	proc 1	proc 2	proc 3
90	E	-	-
1	M	-	-
1	M	-	-
1	M	-	-
90	SM	SC	-
60	SC	SM	-
1	SC	SM	-
60	SC	SM	-
90	SC	SM	SC
60	SC	SC	SM
1	SC	SC	SM
60	SC	SC	SM
515			

Stream 2

MESI

	cycles	proc 1	proc 2	proc 3
r1	90	E	-	-
r2	90	S	S	-
r3	90	S	S	S
w1	1	M	I	I
w2	90	I	M	I
w3	90	I	I	M
r1	90	S	I	S
r2	90	S	S	S
r3	1	S	S	S
w3	1	I	I	M
w1	90	M	I	I
	723			

Dragon

cycles	proc 1	proc 2	proc 3
90	E	-	-
90	SC	SC	-
90	SC	SC	SC
60	SM	SC	SC
60	SC	SM	SC
60	SC	SC	SM
1	SC	SC	SM
1	SC	SC	SM
1	SC	SC	SM
60	SC	SC	SM
60	SM	SC	SC
573			

Stream 3

MESI

	cycles	proc 1	proc 2	proc 3
r1	90	E	-	-
r2	90	S	S	-
r3	90	S	S	S
r3	1	S	S	S
w1	1	M	I	I
w1	1	M	I	I
w1	1	M	I	I
w1	1	M	I	I
w2	90	I	M	I
w3	90	I	I	M
	455			

Dragon

cycles	proc 1	proc 2	proc 3
90	E	-	-
90	SC	SC	-
90	SC	SC	SC
1	SC	SC	SC
60	SM	SC	SC
60	SM	SC	SC
60	SM	SC	SC
60	SM	SC	SC
60	SC	SM	SC
60	SC	SC	SM
631			

There was a little confusion on what the cycle time for an upgrade should be. This solution uses only one cycle for an upgrade and 60 for an update. Answers which use 60 cycles for the upgrade are acceptable too.

For the first stream, the invalidate protocol results in less traffic than the update protocol because other processors do not need the updated data. The extra bandwidth used in sending the updates data is a waste. In the second stream, data which is written by one processor will be used by other processors, therefore the update protocol results in fewer cycles. The invalidate scheme requires each processor to re-fetch the entire line. The third stream shows one processor writing a piece of data several times before any other processor uses it. Here the invalid protocol outperforms the update protocol because consecutive writes to the same location do not need to be seen by other processors, only the latest write.

3) Alternatives for invalidation (H&P 6.6):

The new cache coherence scheme suggested in this problem is to add an additional valid bit per word. In order to understand the complications that arise from this solution to false sharing, we need to redefine what the states of the MSI protocol mean.

When a cache line is brought into the system for reading, all of the valid bits associated with each word are set, and the state of the cache line is in the shared state. If another processor writes to one of these words, it will send an invalid message on the bus and the corresponding valid bit of the word being written to on the other processor is reset. So we can have a condition where some words are valid and some are not. Therefore, all words in this cache line which have their valid bit set are shared. This is similar to the original definition of shared.

However, when the cache line is in the exclusive state, then *one or more* valid words are exclusive to this processor. In the original protocol, when a word in the line was written to, the entire line was invalidated on all the other processors. Since we are now trying to avoid the problems which arise by doing that (for example, false sharing) we only invalidate the words that will be written to. One of the complications that arise is that for future writes, we do not know which words have been invalidated on other caches. The protocol will have to invalidate the same word on all other processors on every write!

Another complication is when a processor needs an update of the contents in a cache line. Multiple processors may have the line in their cache, but each processor may have different words in the line which are valid. Every cache will need to send out the line on the bus for a complete update. This is difficult because there is no way of knowing how many responses to an update to wait for.

4) Invalidate and update cache coherence schemes (H&P 6.7):

This question asks you to write several bits of code that illustrate the pros and cons of invalidation and update cache coherence protocols. We use N to refer to the number of processors, pid to be a variable holding the current processor's ID, and k to be an integer with a fairly large value throughout this solution. Please note that your solutions may be very different yet still correct as long as they show off similar features of the two

protocols. Also, the numbers associated with this problem are basically red herrings — your code is the key.

a) For the first part, we must illustrate the difference in bandwidth.

The first bit of code makes update look much better:

```
for (i=0; i < k; i++)
{
        if(pid == 0) write(X);
        if(pid != 0) read(X);
}
```

In this case, processor 0 is writing a variable X every time through the loop while everyone else reads it. After initial cold-start misses, processors using an update protocol will never miss, as they just receive the updated value of X automatically. As a result, the steady-state bus bandwidth per loop iteration is only one update event (4-8 bytes per loop iteration, plus address overhead). On the other hand, with an invalidation protocol the line containing X will be invalidated out of everyone's cache by processor 0 every time through the loop. They will then all need to re-read in the entire updated cache line from processor 0 or memory, requiring 64 bytes • (N–1) plus address overhead per cycle, minimum. Moral: a small number of writes and many readers makes update work well.

The second bit works better with invalidation:

```
for (i=0; i < k; i++)
{
        if(pid == 0)
        {
                for (j=0; j < k; j++)
                        write(X);
        }

        if(pid == 1) read(X);
}
```

In this case, processor 0 writes a variable X many times before processor 1 reads it once. With an update scheme, this results in k updates of the variable X every time. If k is large, this can add up to a *lot* of bus traffic per cycle. On the other hand, an invalidate scheme just bounces the cache line containing X back and forth between processors 0 and 1 once every cache cycle, requiring about 128 bytes of data bandwidth plus some address overhead each time through the outer loop. Moral: a large number of writes with few subsequent readers favors invalidation.

b) For the next part, we are to write code that shows the potential latency advantage of update. This is best demonstrated by passing updated values down a chain of processors:

```
int chain[k+1];

chain[0] = 1;
chain[ . . . the rest . . . ] = 0;

while(chain[pid] == 0) {}
chain[pid+1] = 1;
```

Here, each processor will wait in the while loop, testing its assigned position in the chain until the previous processor releases it. This release action will propagate down the line of processors one by one. As a result of the testing loop, the assigned portion of the chain variable will definitely be in each processor's cache when its "turn" comes up. With an update protocol, the time required for each step is just *one bus event* — the update from processor i to processor i+1, which will go cache-to-cache. **No** cache misses will be incurred upon the chain variable as the processors pass the release permission down the line. On the other hand, with an invalidation protocol multiple events are required. Processor i must first request the line containing chain[pid+1] in an exclusive state, invalidating it in other processors. After the line is modified, the later processors that are "watching" that line (including processor i+1 and any other processors with a chain array entry on the same cache line that will be affected by false sharing) will miss in their local caches and must subsequently fetch the line from memory or processor i. This will thus require at *least* two bus transactions — one Read Exclusive from processor i and one Read from processor i+1 (answered by a writeback from i, which may require yet another bus transaction) — and will also put a cache miss by each processor in the critical path through the loop. As a result, this loop will take *at least* twice as long on an invalidation-based system, and is likely to take even longer when delays caused by the cache miss and false-sharing effects are taken into account (and which are both conveniently eliminated in an update environment).

c) The second bit of code from part a is actually a good demonstration of how an update protocol may produce a lot of bus contention. However, we will illustrate further with an example of code that will *really* clog up the bus on an update-based system:

```
char databuffer[N+1][k];

for (i=0; i < k; i++)
{
        databuffer[pid+1][i] = 0;
}
for (i=0; i < k; i++)
{
        databuffer[pid][i] = 0;
}
```

This small example performs a fairly innocuous and common task — clearing out a data buffer of char elements between uses — a couple of times in succession. However, it is a disaster on an update-based system. During the first loop, a well-designed update protocol will probably hold all of the databuffer cache lines in an "exclusive" state, avoiding updates for those cache lines. During the second loop, they will definitely become shared between two processors. At this point, *every character written will require an update on the bus*. A loop that requires only a handful of machine instructions now requires a 50-cycle memory access on the system bus just to transmit a single character throughout the system. Even worse, all N processors will be executing similarly bad code *simultaneously*, magnifying the effect further and making each loop iteration take 50•N bus cycles, minimum. Needless to say, this will clog things up to an extraordinary extent. Meanwhile, an invalidate-protocol system can execute this code much more efficiently because it *always* localizes cache lines as they are written and just requires simple cache misses every 64 loop iterations, as the processors start writing a new cache line.