CMPE 110 Homework #4

John Allard
March 10th, 2015
jhallard@ucsc.edu

1. **Question #1 - Cache Coherency** Assume we have the following quad core system : (picture shown in handout).

   The processor is 32-bit and supports 32-bit virtual address and 32-bit physical address. Each core has its own dedicated IL1 and DL1 for instruction and data cache. L1 cacheline size of all the caches are 4-words. L1 is 4-way set associative, with 4KB size. core0 and core 1 share the same L2 cache. core2 and core 3 share the same L2 cache. L2 line size is 16-word, it is a 8-way set associative cache with 64KB size each. Assume the processor implements MESI cache coherence protocol.

   (given in handout)

   Answer table for Question #1 is on the next page

   Question #1 Explanation - Here I will try to summarize and explain why I gave the values I did in the table. To start, the first to accesses are both misses, as neither `0x100` nor `0x150` are in any cache line, so these miss on both L1 and L2 and have to go down to main memory. Once this is done, the C0-L1 cache contains exclusive access to these values, as does the C0-C1 L2 cache. Next, C1 attempts to strore to `0x100`, which turns the C1-L1 value for `0x100` into the modified state, setting the same value in the C0-L1 cache to invalid. The C0-C1 L2 cache maintains that this address is also modified. C1 then tries to store to `0x104`, which is in the same cacheline as `0x100`. This means it does get a hit on both the C1-L1 and the C0-C1 L2 cache. The state value before was modified from when C1 wrote to `0x100`, and after this operation both the L1 and L2 cachelines for C1 will stay in the modified state, while the `0x100` value for the C0 processor becomes invalidated. C1 then loads `0x150`, which it does not have a copy of in L1, so it has an L1 miss. There is a valid copy in the C1 L2 cache though, so that's a hit and the state for `0x150` becomes shared because it has now been read and not modified by both the C0 and C1 processors. The next call loads `0x154` into the C2 L1 cache, this is a miss because all C1/C2 caches are cold at the moment. This miss means the value has to be brough up from memory, and since we loaded a value that is contained inside of the L2 cache for C0/C1 and inside of the C0 L1 cache, this value goes to the shared state after a remote-read request has propogated through the busses. The next instruction comes for C3 and issues a store to `0x100`, which is a value that is contained in some of the other caches. C3-L1 experiences a miss (it's still cold), then a miss from C2-C3 L2 cache, because it hasn't touched this value either. This means it needs to go down to main memory, while also propogating a Remote-Read-Exclusive value along the various busses to let the C0-C1 L2 cache that we mean to write to a value contained in its cache. This means the other values of `0x100` become invalidated, while the value for `0x100` in the C2-C3 L2 cache and the C3-L1 cache becomes modified. C3 then tries to store to `0x150`, which causes a miss on the L1 cache for C3 but a hit on the L2 cache because of the previous load of `0x154` by the C2, which is in the same cacheline as `0x150`. So we obtain a hit in L2, and propagate a remote-read-exclusive request along the various busses, informing them of our wishes to modify this value. This leaves the cacheline containing `0x150` in both the C3 L1 cache and the C2-C3 L2 cache to modifed, and any other copies of this value become invalid. Finally, C1 tries to load the addresses `0xA00100`, `0xB00100`, and `0x100`. Because of the cache structure, these all belong on the same set of caches. The first one is a miss, because there is no hit on the tag in the cache set. This causes us to have to go down to main memory and extract this value, bringing it back up and setting it to the exclusive state for both the C1 L1 and the C0-C1 L2 cache. The same situation happens with the next address, `0xB00100`, we miss in the L1 and L2 cache as this address nor any others near it have been brought up from main memory before. We insert this into the same cache sets as the last address and set the state bit to exclusive, and once again no other caches on the L1 or L2 levels contain this address. The last address, `0x100`, has been seen before but was since invalidated by a store operation to this address by C3. This means we see it is invalid, send a remote read request throughout the busses, and we have to have this value sent down from the L2 cache for C2-C3, as this has the most recent value. Both L2 caches then contain `0x100` in the shared state, as does the C1-L1 cache.

Table 1: Question #1 Calculations

	Core	Instr.	Addr.	L1 Access	L1 Curr-State	L1-L2 B	L2 Curr-State	L2 Access	L2 State	L1 State
1	0	ld	0x100	miss	I	Rrd	I	miss	E	E
2	0	ld	0x150	miss	I	Rrd	I	miss	E	E
3	1	st	0x100	miss	I	Rrx	E	hit	M	M
4	1	st	0x104	hit	M	Rrx	M	hit	M	M
5	1	ld	0x150	miss	I	Rrd	E	hit	S	S
6	2	ld	0x154	miss	I	Rrd	I	miss	S	S
7	3	st	0x100	miss	I	Rrx, Bwb	I	miss	M	M
8	3	st	0x150	miss	I	Rrx, Bwb	S	hit	M	M
9	1	ld	0xA00100	miss	I	Rrd	I	miss	E	E
10	1	ld	0xB00100	miss	E	Rrd	E	miss	E	E
11	1	ld	0x100	miss	E	Rrd	E	miss	E	E

2. **Question #2 - Memory Mapping Impact**

Assume we have the following single threaded code: (Given in the handout)

(a) **Question 2.A - Single-threaded vs Multi-threaded**
Assume data in each of the a, b, and c matrices are stored in the memory in the following way:

Assume the same cache hierarchy as in Question 1. How many data cache misses is the single threaded version to have? How about the multithreaded version? Does the average data memory access change between running the single threaded and multithreaded? Does it matter which thread is mapped to which processor? Justify your answer.

```
addr: a[0][0]
addr+0x4: a[0][1]
...
addr+0x1C: a[0][7]
addr+0x20: a[1][0]
...
addr+0x3C: a[1][7]
...
```

Answer : Cache misses for the single threaded : 42 total (which is 43.75% of the total accesses).
Explanation - I found the misses for the single threaded program by finding a pattern and extrapolating rather then looking at every single of the $3 * 8 * 4 = 96$ seperate memory operations(both reads and writes). First, since the L1 cache cache only store 4 words per line, we are going to get a miss every 4 accesses per variable. So a[0][0], b[0][0], and c[0][0] all miss on both L1 and L2 cache initially, bringing in the values for a[0][0]-a[1][7] (16 words) (for b and c arrays too) into the L2 cache, and loading a/b/c[0][0] through a/b/c[0][3] into the L1 cache. This means that the next accesses for a/b/c[0][1] through a/b/c[0][3] will all hit, before another miss occurs on the L1 cache. What is important to note though is that after this miss we don't need to go to memory, we can just go to the L2 cache to get the next 4 values. This means we miss at the first level but hit at the second, as opposed to when we started where we missed on both L1 and L2 cache. So we know that we miss on L1 cache 1 out of every 4 times we try and read/write a value, because the L1 cache is 4 words long and we have to load from L2 or main memory when we reach an address outside of those 4 words. On top of that, every 16 write/reads for each of the 3 arrays we miss at L2 and have to go to main memory. This means our total number of misses is $\frac{96}{4} + 3 * \frac{96}{16} = 42$. Once again, this counts misses to both the L1 and L2 caches seperately, a miss on the L1 cache and a hit on the L2 cache adds both a hit and a miss to the total calculated above.

Cache misses for the multi threaded : 30 total (which is 31.25% of the total accesses)
Explanation - The multi-threaded has the advantage that a single bring up of data to the L2 cache will reduce the miss rate for both of the L1 caches that rely on that L2 cache. The reason that the number of misses doens't seem to be that much fewer than the single threaded system is because each thread still needs to access 8 words of data for each of the a/b/c arrays, and since each L2 cache can only hold 4 contiguous words of data, each of the cores will miss twice per array. So 4 cores, missing twice for each of the 3 arrays gives us 24 total cache misses at L1. The L2 caches we only miss twice for each array, one time for each L2 cache when they are originally cold. Once they are loaded with the data then there won't be any more misses to the repsective L2 caches. This gives us a total of $\frac{96}{4} + 3 * 2 = 30$ total misses, and relatively good improvement over the single-threaded.

The average memory access time is better for the multi-threaded, this is because it doesn't have to go to main memory to load the L2 caches nearly as often. Going to main memory from the L2 cache is a very expensive operation, so this improves the time over the single threaded version. Also, it does matter what processors are running which threads. I have core n corresponding to thread n, which allows the L2 cache for each pair of L1 caches to contain data that is common to both of them. If I were however, to have C0 run thread 3, C1 run thread 1, C2 run thread 0 and C3 run thread 2, then the L2 cache wouldn't contain overlapping data for both L1 caches above it and this would cause much more frequent misses to the L2 cache.

It is also important to note that the speed-up between the two paradigms is more pronounced then simply looking at the difference in hit rates, because it the multi-threaded version many of the hits and misses are happening at the same time in different cores. So even though the hit rate isn't too much better, we still end up with a significant speed improvement because we aren't waiting to finish c[0] before we start c[1], c[2], and c[3].

(b) **Question 2.B - New Memory Mapping**

If we change the memory mapping as follows, how would it change the cache hit rate? You dont need to compute the exact cache hit rate, but provide enough justification for your answer, particularly from a locality perspective (you can list the data addresses a sample thread touches to guide you). Which version would have a better performance?

```
addr: a[0][0]
addr+0x4: a[1][0]
...
addr+0x1C: a[7][0]
addr+0x20: a[0][1]
...
addr+0x3C: a[7][1]
...
```

Note - This was asked online on Piazza but no teachers got around to answering, so I'm stating my assumption here. I believe the above list to be a mistake, because it shows the arrays being of size 8x4, when they are supposed to be 4x8 as in the first problem. If the arrays were now 8x4 then the code given in problem 2A would not work, so I'm assuming it is a typo and that it should really only go up to 4 on the first index before going up a digit on the second index.

Answer : This setup would reduce the cache hit rate by a non-insignificant amount. This is because we lessen the locality of the data. Now, when we want to pull from array index a[0][0], the L1 cache will be loaded with a[0][0], a[1][0], a[2][0], and a[3][0]. The problem is these aren't accessed sequentially, and they aren't even accessed in the same threads, they are all accessed in different threads. This means of these 4 values loaded into the L1 cache, only the first one can be used. When the next iteration of the loop comes around and we need to access the array a[0][1], this will not be in the L1 cache (although it will be in L2 cache). Same thing happens when we need to go from a[0][1] to a[0][2], that value is no longer close enough to a[0][1] to be brought into the L1 cache previously, and thus a miss will always occur on each L1 access.
So just to be clear, a single thread in the multi threaded program needs access array elements a[n][0] through a[n][7]. In the previous memory arrangement, these were all contiguous in memory and thus to get all 8 values we only needed to pull data into the L1 cache exactly twice for each array. With our new memory arrangement, the values of a[n][0] through a[n][7] are no longer contiguous, and thus each time we need to access the next element we will have to go down to main memory (actually because L2 is 16 words wide, there are situations where the next word needed is in the L2 cache below, but this is not often).

Interestingly enough, if you switched the inner and outer loops with eachother then this would actually run faster, because with the inner and outer loops switched with each other this setup preserves the spacial locality of the accesses much better than the first memory paridigm does.

(c) **Question 2.C - Pipelined Write Hit Path**

Assume you run the program multiple times. Does the cache hit rate changes assuming running only the single threaded program across different runs? How about the multithreaded across different runs? What is the source of variation?

Answer : (I'm assuming the memory structure from problem 2A, not 2B for this problem)
For the single threaded program, there is no speedup to running it over and over besides the fact that the inital states of each cache will not be cold, so we will only see a relatively trivial speedup. With the multi threaded program, we also see a speedup from the non-cold caches in the beginning but because we are using 4 L1 caches and 2 L2 caches, as apposed to the single L1 and L2 cache from the single threaded, we will have much more data preloaded into our caches. This should give us a slightly larger speedup then was discussed in problem A for the multi-vs-single threaded, just because we will see proportionally more intial misses.

(d) **Question 2.D - Reduction Stage**

The provided code computes the absolute difference of two blocks of an image. This is used to find the difference between consecutive frames in video encoding. Eventually the difference of two blocks needs to be converted to a scalar data. The following code snippet shows the code:

```
int reduction ( int **c ) {
int sum = 0;
for( int i = 0; i < 4; i++ ) {
for( int j = 0; j < 8; j++ ) {
sum += c[i][j];
}
}
return sum;
}
void main () {
...
adiff( a, b, c );
int sad = reduction( c ); // Sum of Absolute difference
...
```

How do you suggest parallelizing the reduction function? Ignore all other instructions, and just consider the add operation and the data dependency. Assume the add operation takes 1 cycle. What is the minimum number of cycles to finish the reduction, in theory?

Answer : I would try and make it multi-threaded in a way that is similar to the way we multi-threaded the original code in parts A and B. It would be split into 4 threads that each replace a single value for the variable i in the outer for-loop, and each thread would need its own temporary `sum` variable. They need their own sum variable because that is the common link between the different threads, they all read from different first indices of the c array but then all write to the sum variable on every instruction. If we broke it up into 4 threads with their own sum variables, this conflict would go away and we wouldn't have to worry about constantly changing the state for the sum variable between modified and invalid for the differnent caches, which would slow down our program a ton. At the end, we would need to have a way to take these 4 different sum values and add them together to the final sum, which would then be returned to the user. I think that technically we finish each thread in 8 additions (so 8 cycles), then we would need 3 more adds at the end to compound the values of sum0, sum1, sum2, and sum3 into the final sum value (add sum1 and sum2, then this value and sum3, then this value and sum4, then write to sum). So this would theoretically be accomplishable in 12 cycles, each core executing 8 in parallel with no conflicts then a final 3 in sequence outside of the threads to compound the individual values into the one sum we are looking for.