CS 61C Fall 2014 Discussion 8 – Cache Coherency

MOESI Cache Coherency

With the MOESI concurrency protocol implemented, accesses to cache accesses appear *serializiable*. This means that the result of the parallel cache accesses appear the same as if there were done in serial from one processor in some ordering.

State	Cache up to date?	Memory up to date?	Others have a copy?	Can respond to other's reads?	Can write without changing state?
Modified	Yes	No	No	Yes, Required	Yes
Owned	Yes	Maybe	Maybe	Yes, Optional	No
Exclusive	Yes	Yes	No	Yes, Optional	No
Shared	Yes	Maybe	Maybe	No	No
Invalid	No	Maybe	Maybe	No	No

1. Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache with one cache block and a two cache block memory. Assume the MOESI protocol is used, with write- back caches, write-allocate, and invalidation of other caches on write (instead of updating the value in the other caches).

Time	After Operation	P1 cache state	P2 cache state	Memory @ 0 up to date?	Memory @ 1 up to date?
0	P1: read block 1	Exclusive (1)	Invalid	YES	YES
1	P2: read block 1	Owned (1)	Shared (1)	YES	YES
2	P1: write block 1	Modified (1)	Invalid	YES	NO
3	P2: write block 1	Invalid	Modified (1)	YES	NO
4	P1: read block 0	Exclusive (0)	Modified (1)	YES	NO
5	P2: read block 0	Owned (0)	Shared (0)	YES	YES
6	P1: write block 0	Modified (0)	Invalid	NO	YES
7	P2: read block 0	Owned (0)	Shared (0)	NO	YES
8	P2: write block 0	Invalid	Modified (0)	NO	YES
9	P1: read block 0	Shared (0)	Owned (0)	NO	YES

2. Consider if we run the following two loops in parallel (as two threads on two processors).

```
for(int i = 0; i < N; i += 2) array[i] += 1;
for(int j = 1; j < N; j += 2) array[j] += 2;
```

Would we expect more, less, or the same number of cache misses than if we were to run this serially (assume each processor has its own cache and all data is invalid to start with)?

Possibly more since both are modifying the same cache blocks causing invalidation of each other's blocks.

Concurrency

3. Consider the following function:

```
void transferFunds(struct account *from,
                   struct account *to,
                   long cents)
{
   from->cents -= cents;
   to->cents += cents;
}
```

   a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: if the problem isn't obvious, translate the function into MIPS first)

   Each thread needs to read the "current" value, perform an add/sub, and store a value for from- >cents and to->cents. Two threads could read the same "current" value and the later store essentially erases the other transaction at either line.

   b. How could you fix or avoid these races? Can you do this without hardware support?

   Wrap transferFunds in a critical section, or divide up the accounts array and for loop in a way that you can have separate threads work on different accounts

Summary of General Speed-up Techniques

- Data-level parallelism / SIMD: compute multiple results at a time.
- Thread-level parallelism / OpenMP: have multiple threads doing computations at a time
- I/D cache locality (ie. loop ordering, etc.): maximize cache hits for higher speed.
- Loop unrolling: minimizes for loop overheads.
- Cache blocking: increase cache usage for higher performance.
- Code optimization (mostly compiler's job): interweave independent instructions to avoid CPU stalls (waiting for the results from the previous instruction).