CS 61C Spring 2016 Discussion 10 – 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				
2	P1: write block 1				
3	P2: write block 1				
4	P1: read block 0				
5	P2: read block 0				
6	P1: write block 0				
7	P2: read block 0				
8	P2: write block 0				
9	P1: read block 0				

Concurrency

2. 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)

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

Thread Level Parallelism

```
#pragma omp parallelism
{
        /* code here */
}
```

*Each thread runs a copy of code within the block
*Thread scheduling is non-deterministic

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
        /* code here */
}
```

Same as:

```
#pragma omp parallel
{
      #pragma  omp for
      for (int i =0; i < n; i++) {...}
}
```

1. For the following snippets of code below, circle one of the following to indicate what issue, if any, the code will experience. Then provide a short justification. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume arr is an int array with length n.

a)

```
// Set element i of arr to i
#pragma omp parallel
(int i = 0; i < n; i++)
        arr[i] = i;
```

Sometimes incorrect   Always incorrect   Slower than serial   Faster than serial

b)

```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
        arr[i] = arr[i-1] + arr[i - 2];
```

Sometimes incorrect   Always incorrect   Slower than serial   Faster than serial

c)

```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
        arr[i] = 0;
```

Sometimes incorrect   Always incorrect   Slower than serial   Faster than serial