CS 61C Spring 2017 Discussion 10

MSI Cache Coherency

MSI is a simple concurrency protocol to prevent write-back caches from maintaining incorrect data. Its approach is to invalidate blocks other caches on write (instead of updating the value in the other caches).

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
Shared	Yes	Maybe	Maybe	Yes, Optional	No
Invalid	No	Maybe	N/A	No	No

1. How many bits do we need to add to each cache block to implement MSI?

2. Consider the following access pattern on a two-processor system with a memory size equivalent to two cache blocks. Each processor has a direct-mapped, write-back, write-allocate cache with only one cache block. Fill in the state of each cache, and in parentheses write the memory block contained in that cache.

Time	After Operation	P1 cache state	P2 cache state	Memory @ 0 up to date?	Memory @ 1 up to date?
0	P1: read block 1	Shared (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				

Concurrency

1. 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 */
}

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

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

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

2. Consider the following code:

```
// Decrements element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel  {
        int threadCount = omp_get_num_threads();
        int myThread = omp_get_thread_num();
        for (int i = 0; i < n; i++) {
                if (i % threadCount == myThread)
                        arr[i] *= arr[i];
        }
}
```
What potential issue can arise from this code?

Data Level Parallelism

__m128i _mm_load1_si128( )	returns 128-bit one vector
__m128i _mm_loadu_si128(__m128i *p)	returns 128-bit vector stored at pointer p
__m128i _mm_mul_ps(__m128 a, __m128 b)	returns vector (a0*b0, a1*b1, a2*b2, a3*b3)
void _mm_storeu_si128(__m128i *p, __m128i a)	stores 128-bit vector a at pointer p

1. Implement the following function, which returns the sum of two arrays:

```
static int product_naive(int n, int *a) {
        int product = 1;
        for (int i = 0; i < n; i++) {
                product *= a[i];
        }
        return product;
}
static int product_vectorized(int n, int *a)  {
         int result[4];
        __m128i prod_v = _____________________;

        for (int i = 0; i < _____; i += ___) {   // Vectorised loop
                prod_v =_____________________________________________;
        }
        __mm_storeu_si128(_____________________, _____________________);

        for (int i = ______; i < ____; i++) {  // Handle tail case
                result[0] *= _________;
        }
        return ____________________________________________;
}
```