CS533 HW2

Abdul Dakkak

Question 1: Prefetching

```
(1) for i = 0 to 127
(2)     for j = 0 to 31
(3)         A[i,j] = B[i] * C[j]
```

a. You are going to get a cache miss $3 * (128 * 32 * 4)/16 = 3 * 1024$ times, so the number of cycles wasted due to cache misses is $3 * 1024 * 64$ cycles $= 3 * 2^1 6$ cycles. So the above loop would take

$$128 * 32 * 16(\text{cyles when their is a cache hit})+$$
$$3 * 2^1 6(\text{cycles for memory fetch due to cache miss})+$$
$$129 * 33 * n\text{where } n \text{ is however many cycles to execute the branch and conditionals}$$

b. It is clear the we need to prefetch data 4 elements away for each aray (e.g. `C[j+4]`) since the cache line is 16 bytes and each element in the array is 4. We can prefetch the `B` array in the outer loop (since it is only dependent on the `i` induction variable), since we will have a cold miss in the first loop iteration, we can service that prefetch before 2a and 2b for the next `i` iteration. After the cold miss, we prefetch `C` and `A` for the next `j` iteration.

```
(1) for i = 0 to 127
(1p)    PREFETCH(&B[i+4])
(2)     for j = 0 to 31
(3)         A[i,j] = B[i] * C[j]
(2a)        PREFETCH(&C[j+4])
(2a)        PREFETCH(&A[i,j+4])
```

Question 2 : Prefetching 2

a. Prefetching tackles the problem of cold cache misses

b. In a single issue processor you can only have one instruction in flight, so a prefetch latency would not be hidden in place of a cold miss. In a multiple issue processor, you can hide the prefetch latency.

c. When you are in the critical section, otherwise you cannot assume that the memory location (which you've now bound a register to) has not been modified by another processor.

d. No, since once you have the prefetched data in cache then the cache coherency model would work the same. Care must be taken however to make sure that you invalidate before you write, otherwise you can prefetch a memory location while another processor is invalidating and writing to it. In binding prefetches, one adds special logic for this issue.

e. In a long latency cache miss enter runahead mode. In runahead mode, speculatively execute instructions to generate prefetches. When original miss returns, you exit runahead mode. The intuition is that this generates some accurate prefetch information and would hide some of the latency of the original cache miss.

f. One can use the cores as memory prefetch units. So you offload some of the loads and stores onto them and use them as a closer memory fetch unit (and their L1 cache as a slightly farther away L1 cache).

Question 3 : Synchronization 1

a. This would change the semantic of the algorithm since the check `bar_name.counter == p` is currently within the atomic block. To see the problem consider the interleaving

```
Thread 1                             Thread 2
UNLOCK(bar_name.lock);
                                     LOCK(bar_name.lock);
                                     mycount = bar_name.counter++;
if (bar_name.counter == p)
```

`Thread 1`'s condition would fail and would go into an infinite loop.

b. Yes

```
BARRIER(bar_name, p) {
    local_sense = !local_sense
    if fetch_and_inc(&bar_name.counter) == p {
        bar_name.counter = 0
        bar_name.flag = local_sense
    } else {
        while (bar_name.flag != local_sense) {}
    }
}
```

c. You still need locks, since different threads might enter the same critical sections if the OS performs context switching inside the section. Consider the following timeline

```
                               Thread 1        Thread 2
> start critical section         i1
                                 i2 (long latency)
> os decides to context switch                  i1
                                                i2
```

this is clearly wrong. And to implement this you would still need an atomic operation (again to guarantee that the instruction sequence get executed in order).

d. `compare_and_swap` is implemented as:

```
compare_and_swap(p, old, new) {
    BEGIN_ATOMIC()
    if(*p == old)
        *p = new
    END_ATOMIC()
    return old;
}
```

Using `LL` and `SC` you can implement `compare_and_swap` as

```
compare_and_swap(p, old, new) {
    do {
        x = LL(p)
    } while(!SC(x == old ? old : new, p))
    return old;
}
```

`fetch_and_inc` is implemented as

```
fetch_and_inc(loc) {
    BEGIN_ATOMIC()
    x = *old
    *loc = x + 1
    END_ATOMIC()
    return x;
}
```

Using LL and SC you can implement fetch_and_inc as

```
fetch_and_inc(loc) {
    do {
        x = LL(loc)
        y = x + 1
    } while(!SC(loc, y))
    return x;
}
```

Question 4 : Synchronization 2

a. Here CH is a cache hit, TS is the test and set operation, and WR is write back.

```
p1     p2   ... p8
INV    INV      INV
RM     RM       RM
TS     CH       CH
WR     RM       RM
       RM       RM
       TS       CH
       WR       RM
            ... TS
```

From the above, we can figure out that there is $3 + 4 + 5 + ... + (8 + 3) = 60$ bus transactions.

b. There is no guarantee that the first processor would get serviced first, so since there are 60 bus transactions then 300 cycles elapse before all 8 processors get serviced. So, on average $P1$ would take $\frac{300}{8} = 37.5$ cycles.

c. If there is no cache then test-test-and-set would still generate less traffic than test-and-set, since test-and-set is essentially an atomic read/write while in test-test-and-set we can perform the read without the need for the write in many iterations. So, you'd save a bit less traffic by not sending data into memory, and just reading from it.

d. Yes, recall that one can implement either using LL and SC (previous question).

Question 5 : CMT and SMT

a. SMT differs from superscalers by providing parallelism at the thread level rather than at the instruction level, and it different from traditional multithreading by using the same core to execute the instructions from different threads simultaneously.

b. The structures

- Branch Predictor — Augmented with a thread id for SMT. By augmenting with the thread id, you'd increase your prediction accuracy
- Return Address Stack — Duplicate

- Register File — Shared
- TLB — Shared
- RAT — Shared
- LSQ — Shared

c. As you increase your issue width you will increase the rename, wakeup, and bypass delay. This is due to the increase in wire length.

d. For CMP these are traditional independent cores with a coherence model. Some hardware is also needed for CMP to allow it to communicate registers with other processors. SMT requires one to duplicate certain hardware structures, such as the PC, return address stack, etc... this increases the cost and latency of certain operations. The advantage of SMT is that you can avoid some structural hazards that you'd otherwise get in a uniprocessor machine.

e. A task may get squashed either because its predecessor gets squashed or if detects that a load and a store conflicts (did not happen in program order). Subsequent tasks also get squashed.