CSC458 HW2

Chi-Chun, Chen

Ex 1

1-1

Question: An INVALIDATE coherence action in an inclusive cache hierarchy needs to be sent only to the shared cache.

True: From the property of inclusive multi-level cache hierarchies, we know that if level-n cache has been updated, then all the level greater than n are also updated. Also, if the cache hierarchy has the inclusion property, the invalidation flushes the upper level cache if necessary. Meaning, if one of the core invalidate a shared cache, then the shared cache of other core invalidate the upper level cache. Hence, the **Invalidation** needs to be sent to the shared cache only.

1-2

Question: Snoop-based cache coherence protocols are easier to implement on systems with exclusive caches.

False: To answer this question, I assume that not all the level of cache have independent bus snooping hardware. The assumption is made from the idea of PCA[1] that if L-1 cache and L-2 cache both have its own bus then there will be lots of duplicate work, in addition, L-1 cache is on-chip cache so it might not worth a precious pin to have bus on L-1. To make the assumption more specific, I assume that the hardware has two level of cache and only level-two cache is connected to the bus. Back to the question, the answer is **False**, comparing to inclusive cache hierarchy, if one of the core issue a write request and update its L-1 cache, the snoop bus (in level-2 cache) won't aware the update to L-1 cache, therefore, an addition implementation of update L-2 cache is necessary.

Ex 2

2-1

Question: True or false: At optimization level -O3, gcc 4.8 will eliminate the while loop above.

True.

2-2

Change the definition of int *a to volatile int *a. Will the loop still be eliminated? Explain.

Answer: The loop won't eliminated by the compiler if change from $int * a = \&b$ to $volatileint * a = \&b$. Since the decorator "volatile" in **C programming language** means that every update of the variable is not stored in register, instead, all of the update is stored back to main memory. If the update is stored back to main memory, then the compiler cannot guarantee if the variable is modified by other threads, hence, it cannot eliminate the loop.

Ex 3

The reason why there are no fences in the resulting assembly code when compiled with gcc on an **x86** architecture is that **x86** do not reorder atomic instructions with load and store[2]. However, the **Power** architecture do reorder atomic instructions with load and store[2]. Therefore, fence instructions is needed to make sure that the resulting assembly code is not reordered.

Ex 4

To answer the question, it is easier to discuss the fence in **lock.acquire** and **lock.release** separately. For the fence at the end of **lock.acquire**, it guarantees that the lock is held before the thread can execute any instruction in the critical section. For the fence at the end of **lock.release**, it makes sure that all the work in the critical section has to be done before the lock is given to other thread. To be more specific, every load and store in the critical section must happen in the critical section and the fence in the end of **lock.acquire** and **lock.release** explicitly command that compiler could not put instructions inside the critical section outside the critical section.

It's clearer to explain with an code snippet below. $fence(R||RW)$ is the last statement in **lock.acquire**, the fence on the read before read/write is for ensuring that the read of the lock variable has actually been read before acquire is done. And the read/write after the read is for making sure that the instructions of $a++$ cannot occur before the thread has acquired the lock. $fence(RW||)$ in **lock.release** is to guarantee that all of the instructions in the critical section, namely, instructions of $a++$, must be done before leaving the critical section, which is when the lock has been successfully released.

```
1 // a is a global variable shared by all threads
2 int a = 0;
3
4 // statements in threads
5 lock.acquire();
6 // inside acquire/release is a critical section (Mutual Exclusive)
7 a++;
8 lock.release();
```

References

[1] Parallel Computer Architecture: A Hardware/Software Approach

[2] Memory Ordering Relaxation on Processors
https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaaw/ordering.2006.03.13a.