EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 4: September 6

Lecturer: Josep Torrella, Dept of Computer Science, UIUC *Scribe: Wenwen Zhang*

4.1 Preface

With the advancement in integrated circuits, we have accelerated progress in transistor integration. We can create new advancements such as large multicore processors for data centers and cloud, and 3D stacked chips. With these new advancements, we also encounter a huge power wall, in such supercomputers would take a lot of power input, even in idle states. Power consumption advancements are relatively slow in comparison to technology improvements, which makes computer architecture improvement high priority. In order to resolve this problem, computer architects are focusing on improving energy efficiency and faster communication/synchronization between computers. In this seminar, Dr. Josep Torrella focuses mostly on new technologies to reduce the cost of basic primitive for parallelism and other challenges in energy and programmability.

4.2 Goal

In order to make synchronization less expensive, we can improve in three different areas.
- make memory fences less costly/free
- break serialization in lock-free synchronisation
- create scalable concurrent priority queues

4.3 Memory Fences

Fir the first suggestion to make synchronization inexpensive, we can improve memory fences. Memory fence is a primitive for parallelism such that it can prevent the computer and hardware from reordering memory accesses inserted by programmers or compilers. It does so by forcing both read and write instructions to be finished and retired from the pipeline before the next instruction. Fences can be achieved by programmers inserting codes with fine-grain sharing then compiler insert fence after the access and not reorder code.

Example for memory fences would be as follows:
Executing the following code:

```
1. A0: x = 1;
2. B0: y = 1;
3. A1: t0 = y;
4. B1: t1 = x;
```

Incorrect execution without fence could have possible execution order such as A1, B0, B1, A0, which would

result in wrong values for t0 and t1, a demonstration of Sequential Consistency (SC) Violation. If we can insert fences after A0 and B0 to force execution of A0 and B0 before A1 and B1.

4.4 WeeFence (WFence)

Modern implementation of fences at hardware scale could be performing speculations on read instructions after fences. In this case, if no processor observes it, no problem would be caused; if coherence transaction received, this rd would be squashed and retired. However, with speculation, the reads are still not able to retire until the WB is drained. This could still make fences costly.

WeeFence is then introduced with a goal to eliminate any stalls in the pipeline. With WeeFence, post-fence read can retire before the pre-fence writes have drained and ?skip? the fence. WeeFence would only stall when a read that violates SC and detects SC by using a global Pending Set (PS) table to store instructions that passes fences.

4.5 Asymmetric Fences

WeeFence has a pretty good improvement in preventing stalls since Cycles that break SC rarely happens. However, a global PS table is rather expensive to maintain. Without the global PS table, deadlocks can occur when all the processors stall themselves after encountering other fenced instructions. Only exception is when one process stall right before the fence. Thus, creating this exception could be the key to resolve the problem.

We can create this exception by having a Strong fence and N-1 Weak fences for a given conflict cycle of N processors. Strong fences are the conventional fences and weak fences are fences like WFence. This can potentially reduce the cost of a global PS table to just a conventional fence. If the strong fence is placed somewhere in the code that less likely to occur, the cost of this strong fence can be further reduced. With implementation of WFence, 90% of fence stall time were eliminated and the overhead of supporting SC went down from 40% to 2%.

4.6 Breaking Serialization

Another bottleneck for increasing power consumption is that many processors have to synchronize on the same variable while executing multiple functions. This case occurs in many areas, such as in OS, databases, language runtimes, and memory allocators. One possible and common solution is to use Compare-and-Swap (CAS), a lock-free synchronization method that uses atomic instructions to manipulate data. However, this could waste a lot of time on processors since only one can set the variable at a time.

In order to reduce the time that other processors have to wait on setting the shared variable using CAS, we could use CASPAR. CASPAR can be done in two steps: first, put the old requests in hardware in queue to allow only one CAS operation at a time; then an eager forwarding method that caches the variable to other processes to prevent stalling. New values after the CAS operation then can be compared with the cache value to ensure accuracy.

4.7 Scalable Concurrent Priority Queues

Since some algorithms work best if some parallel tasks executed before others, we could potentially improve their executing priority to make synchronizations less expensive. However, traditional priority queue always dequeue from head and CASPAR won?t work since it needs to insert nodes like a stack. If all dequeues happen on the highest priority node, it would lead to contention at the head node. In addition, the highest priority node may not be the global best to be dequeued.

An implementation of the new priority queue would be a mix of hardware and software to create and organize sub-queues in different processes to decrease the necessary contention to the high priority node. Hardware collects the nodes at the head of all the queues and sort them to provide one of the top nodes chosen from the highest range. This process resolves the contention to the highest priority item by choosing other items with relatively high priority often. In addition, this implementation reduces 64-threaded application execution time by 2 to 5 times on average.

4.8 Conclusion

In the end, Dr. Torrella briefly talked about other project that he has worked on in the field of processor technology. For example, WiSync, On-Chip Wireless communication concept to include antenna within microchips that would bring wireless network within wired network to improve communication; QuickRec, a prototype of record and Replay (RnR); ScalCore, a core for voltage scalability. With the techniques mentioned above, Dr. Torrella believe that there are a lot of room to innovate in computer architecture field at this time as many exciting interdisciplinary venues of research happens to increase performance, energy-efficiency & programmability.