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

Lecture 4: September 6

Lecturer: Josep Torrellas *Scribe: Brad VanderWilp*

4.1 Introduction

As technology advances, we are able to build bigger and better computers. More, smaller transistors, larger cloud cores, and 3D stacked chips allow us more processing power is computers. The need to utilize this processing power has pushed research towards many-core computers. However, difficulties arise as power consumption increases and the rate of electrical improvements slows. Thus, research prioritizes energy efficiency, fast communication, and ease of programming as we explore the possibilities of parallelism.

4.2 Memory Fences

One way to improve synchronization is through the use of fences. "Fence" is a primitive used by both programmers and compilers to prevent the reordering of memory accesses. For example, in the sequence $write - x, fence, read - y$, the effects on the read on y cannot be observed by any other process until after the previous write has retired from the pipeline and the write buffer (WB) has been drained.

The following illustrates the effectiveness of fences. Consider the following code:

Process1 $A0 : x = 1; fence; A1 : t0 = y$

Process2 $B0 : y = 1; fence; A1 : t1 = x$

Here, fences prevent the order of retirement from being $A1, B0, B1, A0$. This is a possibility because of the time memory can take to propagate. This shows fences ability to preserve Sequential Consistency (SC).

4.2.1 Work-stealing Algorithm

The order between accesses enforced by fences can be used in work-stealing algorithm. Here, several worker processes are given a queue of tasks, which they execute from the tail-end. If a worker completes all of its tasks, it proceeds to steal a task from the head of a different worker's queue. To avoid interfering with each others' task lists, fences can be where workers and thieves perform their check whether the head of a queue is also the tail. A fence is placed between the thief's *get* of the head and check of the tail, and fence is placed between the worker's *get* of the tail and check of the head

4.2.2 Past implementations

A simple implementation for fences would be to stall all memory operations at a fence until other instructions have retired. This, however, leads to a lot of time wasted waiting. Another option would be to allow data to be loaded speculatively after a fence. If a read after a fence is observed through the cache of another

process, the read is squashed. This implementation also prevents reads from retiring until after the WB is drained. Still, there is a need for yet cheaper fence implementations

4.2.3 WeeFence

WeeFence (WFence) is a proposed type of fence that is skipped and allows statements to execute even before the WB drains. This prevents write misses from piling up before a fence. The only stall that needs to occur is right before a read that would violate SC. This is detected by using a global pending set (PS) table. Before a fence is passed, statements are pushed onto the table. If an instruction occurs after a fence that would violate SC based on some statement in the table, a stall is issues.

This method is successful partially because cycles that break SC are generally uncommon. Additionally, WFence is convenient because no compiler support is required. However, maintaining this global table is expensive. Without a global state to order the PS, deadlock could occur as processors stall for each others' fenced statements. But, if at least one processor stalls before the fence, the deadlock is broken

4.2.4 Asymmetric fences

The asymmetric implementation of WFences uses 1 conventional "strong" fence and N-1 "weak", WFence-type fences for a given conflict cycle of N processors. This eliminates the cost of maintaining a global table at the cost of keeping a slower, strong fence. This can be optimized by strategically placing strong fences at sections of code that occur less frequently, such as at initialization rather than in the loop that uses a variable. In the previously mentioned work-stealing algorithm, for example, the strong fence would be used in the thief code because it is the less common case.

Overall, the use of WFences has been shown to be effective. Tests have revealed WFence to eliminate ninety percent of fence stall time and reduce overhead from forty percent to two percent when compared to normal fences.

4.3 Breaking Serialization

Another major objective for parallelization is to alleviate the bottleneck caused by multiple functions accessing the same variable. One common answer to this problem is to use Compare-and-Set (CAS) algorithms. These operate by obtaining the value, changing it, and then performing an atomic operation to set the value only if the variable has not been altered by a different process. However, this leads to processors wasting time, as only one sets the variable at a time.

4.3.1 CASPAR

A modern implementation attempts to eliminate the time that a process must wait to change and use a shared variable with CAS. First, load requests for the old variable value are queued. This allows only one CAS operation to be performed at a time for completely serial execution.

The next step is an eager forwarding method. The cache containing the variable is forwarded between processes. This prevents stalling as execution occurs in parallel. The new value of the variable is later compared to the actual value.

4.4 Scalable Concurrent Priority Queues

One other goal for improving concurrency is improving implementations of priority queue. Some algorithms work best if certain parallel tasks are executed before others, necessitating an efficient priority queue. However, significant overhead can arise when many processes attempt to simultaneously access the highest priority item. One design involves relaxing the priority and giving different processes sub-queues. However, this potentially leaves the optimum of the sub-queue to be far from the global optimum.

Instead, one implementation uses a mix of hardware and software to organize sub-queues. A hardware register tracks the head of all sub-queues, and all items are enqueued to the local sub-queue. Upon dequeueing an item, all subqueues are visited and sorted. Then, one of the items in the highest range is chosen. By not choosing the highest priority item every time, conflicts between processes are avoided. Such implementations have been found to produce 2-5x speedup.

4.5 Other Projects

Optimizing the usage of growing processor technology is a multidisciplinary task which include many other projects than the ones discussed here. For example, WiSync is a concept which includes antennae within microchips. This improves communication by create a wireless network within a wired network. QuickRec is a system of modifying the cache hierarchy to record the non-deterministic events that occur during execution of a program. This provides a useful debugging tool for parallel systems by using these logs to replicate an execution of a program. ScalCore is a design that generates multiple voltage domains for memory and logic to increase throughput. Since logical data can be transmitted with lower voltage, faster memory accesses can be sent simultaneously, improving speed and conserving energy. Finally, research in control theory has been working to improve controllers for power performance, temperature, and other parameters in such a way as to prevent inefficient competition.