MPCS 52060: Parallel Programming Assignment 2

Justin Cohler

April 16, 2019

SAQ 1

Explain how a queue, implemented in hardware in the CPU, could be used to improve the performance of a write-through cache.

The guarantee of a write-through cache is that Main Memory will always have a correct snapshot of operations, across CPUs. An efficient implementation could decide that all hardware threads place messages onto a queue. A queue's first-in-first-out design ensures that processors will place requests on the queue in order, and that the writer to memory, which reads from the queue, will read data to write in order. Since there is no notion of stale data in main memory in this design, the "M" (Modified) parameter in MESI protocol would not be necessary, since main memory would always have the canonical memory state.

This would be more efficient than MESI in the case where we had lots of processors reading and updating the same data. In this case, we would frequently have "Modified" states across threads, requiring dumps to main memory before fulfilling requests. A write-through cache would not need these intermediary steps in these cases, and would streamline writes and reads from main memory.

SAQ 2

(B.1, Exercise 223) In the MESI cache-coherence protocol, what is the advantage of distinguishing between exclusive and modified modes? What is the advantage of distinguishing between exclusive and shared modes?

The distinction between "Exclusive" and "Modified" ensures that modified data writes back to main memory. From section 18.4.1, when a processor broadcasts a request to load a line in Exclusive mode on the bus, other processors will snoop that request. Any other processor with a Modified copy of that line will force a write to main memory before the requesting processor can load the line.

This choreography ensures that request to memory does not provide outdated/stale information to the requesting processor by forcing an update from the processor which modified the line in question.

SAQ 3

(B.1, Exercise 220) Imagine you are a lawyer, paid to make the best case you can for a particular point of view. How would you argue the following claim: if context switches took negligible time, then processors would not need caches, at least for applications that encompass large numbers of threads. Also, critique your argument.

In a world where throughput is king, enemy number one is CPU downtime, where CPUs must wait for memory to be retrieved, and thus do not have operations to perform.

A round-trip to main memory for information takes on the order of about 100 cycles, which is incredibly slow. However, consider a system with tens, or even hundreds of threads and instantaneous (approximately zero cycles) context switches. In this case, a processor could send out one request for state and data info to main memory per thread, and then context switch to each appropriate response as they came in. Assuming that processing instructions for each returned piece of memory took several cycles, the processor would be under constant load. Each time it exhausted an instruction set for a given piece of memory, it would instantaneously context switch, retrieving the next piece of memory and thread state from the bus.

In the world described, the processor would never need caches, because the speed of retrieval is not the limiting factor. The processor will not have downtime waiting for responses from main memory, because with enough threads, there will be a steady stream of work from each thread to switch between.

Critique:

The obvious flaw in this argument is that throughput concerns could be mitigated in a world of high thread-counts and zero-context switch time, but latency would be much worse without the existence of caches.

Imagine a system with a high number of threads, but very little actual data coming in. Amazon's servers, for example, might be very busy on Black Friday but less busy in January. While there are many threads, there are very few requests to main memory. In this situation, without caches, every request would require a slow roundtrip to main memory, and latency would be much slower (100s of cycles) than it would be with caches (10-30 cycles).

SAQ 4

Notice when executing the hw2 preliminaries.go program multiple times, goroutine ids may print in a different order. Why are ids are printing out of order when executing the program multiple times?

Lightweight threads are asynchronous by their nature. Each thread is a separate task, which is sent to a user-level scheduler, which maps these tasks onto threads, which then maps these threads onto processors. Scheduling is done via a *greedy* algorithm, and does not guarantee ordering of schedules.

Moreover, each scheduled task may require reads from various levels of memory (L1-L3 caches, or worst case Main Memory), each of which will take slightly different amounts of time to return. While waiting for memory to be retrieved for each task, the scheduler will prioritize other tasks. This way, throughput is maximized across threads, though a thread may need to wait for the scheduler to come back around to it, which may take slightly longer than if the task was the sole operation on the processor.