Discussion #9: Thread-Level Parallelism

Written by Justin Hsia (7/22/2011)

Thread-Level Parallelism

Thread-level parallelism is what most people think of when they hear the word "parallelism." Having multiple processors at your disposal will allows you to execute many tasks at once. But much like the old adage goes "too many cooks in the kitchen," having multiple processors running at once raises a whole new set of problems.

For the purposes of this discussion, we will assume we are dealing with a *shared memory* system – that is, all the processors operate on the same address space. This will allow us to use our multiple processors to run parts of the same program simultaneously. It is usually safe to assume that each processor has its own cache. If all processors shared the same cache, then two threads working on different parts of memory could cause an unnecessary amount of cache block replacements.

Data Concurrency

Concurrency is just the property of having multiple computations executing simultaneously, potentially interacting with each other[1]. Data concurrency causes problems when multiple processors are trying to access the same piece of data. A data race occurs when an unexpected ("wrong") result can appear based on the exact sequence/timing of the data accesses (reads/writes)[2].

A simple example: `arr[0] = 0` and Thread0 wants to add 5 to it and Thread1 wants to add 10 to it. The expected result is `arr[0] = 15` but if both threads read `arr[0]` before either threads writes back to it, then both threads will think that `arr[0] = 0` before performing their respective arithmetic, so the result will be either `arr[0] = 5` or `arr[0] = 10`, depending on which thread writes back to `arr[0]` *later* (convince yourself why that is).

Data Locks

So what can we do? A nifty idea is that of a *data lock*. A **lock** is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution[3]. Basically, we allow access to a piece of data (which could be a single variable, a struct, an array, etc.) ONLY to the thread that holds the *key* for that piece of data. Each key is itself just a variable that indicates whether it is in use or not. There are multiple schemes for doing this such as lock/unlock and reader-writer lock. But wait! As another piece of data, keys themselves are subject to data concurrency issues! That is, multiple threads can read the lock as unlocked and think that they each now own the key, which defeats the purpose of the lock! We solve this problem with some hardware help:

---

[1] http://en.wikipedia.org/wiki/Concurrency_%28computer_science%29
[2] http://en.wikipedia.org/wiki/Data_race
[3] http://en.wikipedia.org/wiki/Lock_%28computer_science%29

MIPS hardware-enable atomic instructions

One way to prevent the data races is to use *atomic operations*. These operations appear to the rest of the system to occur instantaneously[4] (are uninterruptible). These must be implemented with some help from hardware. These usually occur in a succeed-or-fail manner. That is, before writing to memory, we check to see if the value has changed. If it has not, then we modify the value and report a success. If it has changed, then it means another thread modified the value in the meantime, so we report a failure and do NOT modify the value.

In MIPS, we achieve this using the two special instructions `ll` ("load linked") and `sc` ("store conditional"). `ll` is used for the initial read of the address and then `sc` is used to attempt to write to it. `sc` will read the value of the address again, compare it against the value that was read by `ll`, and override the register that contained the new value with the succeed-or-fail result. The simplest locking mechanism implemented in MIPS functions is shown below. The key either contains the value of 0 if unlocked and 1 if locked. If locked, the other threads will simply wait for the key to become unlocked again:

```
# assume $s0 -> address of key
lock:
      ll   $t0, 0($s0)        # read current value of key
      bne  $t0, $0, lock      # if locked, keep looping
      addi $t1, $0, 1         # if not locked,
      sc   $t1, 0($s0)        #    attempt to lock by storing 1
      beq  $t1, $0, lock      # if lock attempt failed, keep looping

# OPERATIONS ON LOCKED DATA

unlock:
      sw   $0,  0($s0)        # as the only thread holding the key, can
                              #    unlock in a non-atomic fashion.
                              # this will not always be the case in more
                              #    complicated locking schemes.
```

The one confusing thing is that on an operation like `sc $t1, 0($s0)`, whatever value you were attempting to store at `0($s0)` is LOST and overridden with a 0 for store fail or 1 for store success.

Cache Coherence

As we know, cache use is desirable because of the improvement in speed over accessing main memory. But with multiple processors and multiple caches, we need to make sure the data is consistent in each local cache, an issue known as *cache coherence* (or *cache coherency*), otherwise two different versions of the same data may exist in the system.

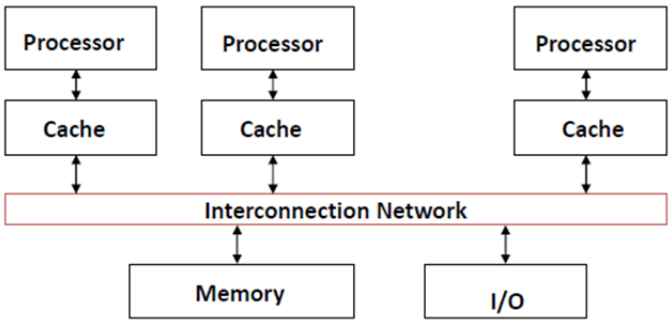
Cache coherence is a tricky subject to introduce quickly because there are many different coherence schemes and protocols (see Wikipedia[5] for a quick overview) and so much

[4] http://en.wikipedia.org/wiki/Atomic_instruction
[5] http://en.wikipedia.org/wiki/Cache_coherence

of it is implementation-dependent. I will attempt to just touch on the broad concepts and we will focus on the *snooping* mechanism and the *MOESI protocol*. Do not forget that the general cache schemes of write-back/write-through and write allocate/no-write allocate are choices that will affect our cache coherence scheme. Here the use of the Valid and Dirty bits in each cache row also becomes increasingly important.

What is the main goal of a cache coherence scheme? To ensure that no cache incorrectly uses an outdated value of memory.

Example Cache Coherence Scheme (MOESI)



As seen in the figure above, the local caches communicate with each other via an *Interconnection Network*. Here we focus on snooping mechanisms, where this communication includes invalidation requests, monitoring of memory requests, and even transfer of data (cache blocks).

Why would you ever need to directly transfer a block from one cache to another over the interconnection network? In a write-back scheme, a local cache can hold a more updated version of a block than main memory. Transferring the block over the interconnection network skips over the lengthy steps of writing the block back to memory for the other cache to read from main memory.

In most cache coherence protocols, each block within a cache is assigned a *protocol state* that indicates a number of things including its up-to-dateness relative to main memory and the other caches, the ability to write to the block, and the cache's responsibility on the interconnection network. The protocols are named by acronyms of the protocol state names, so the MOESI protocol uses the states **M**odified, **O**wned, **E**xclusive, **S**hared, and **I**nvalid. MOESI protocol is basically only used with write-back schemes.

What do the different protocol states mean? I will refer to you the table at the top of Worksheet 9 and Wikipedia[6] for the specifics, but let's go over the general ideas:

Modified and Owned indicate "ownership" of the block in the sense that the cache intends to write to the block without updating main memory. When another cache requests that block, the first cache has the most updated version, so it must respond to the request.

[6] http://en.wikipedia.org/wiki/MOESI_protocol

Modified and Exclusive indicate sole possession of that block while Owned and Shared mean multiple caches have a copy of the block. There can simultaneously be many Shared versions of a block but at most one Owned version of that block (though all must be up-to-date).

Invalid means the version of the block that cache holds is out-of-date and cannot be used.

Any protocol state can transition to Invalid at the request of another cache, and an Invalid state can transition to Exclusive or Shared on a read or to Modified on a write.

What happens when a cache writes to a block? Now that cache has the most up-to-date version of that block. If no other cache holds that block, then nothing need be done (remember we're using write-back). If other caches hold that block, then the two options are to update their blocks or invalidate them. Updating their blocks will involve either writing to main memory for them to read (which is slow), or sending them the block via the interconnection network (which is bandwidth-limited and does not scale up well). So the usual choice is to just have the other caches invalidate their own copies.

False Sharing

This is a concept that results in unexpected reduction in performance. You should have been exposed to it during Lab08, which also references Wikipedia[7]. This problem only arises in thread-parallel systems where each processor has its own cache. A good design principle to avoid this problem is to split up parallel processes spatially.

[7] http://en.wikipedia.org/wiki/False_sharing