George Beuhler
Scribe Notes
Lecture 3/1/12

Locks

Performance

An algorithm's performance is critical to wholly evaluating their effectiveness. If an algorithm performs slowly than it is useless in most practical situations. The bakery algorithm is an example of this; it serves as an academic example but performs too slowly to be effective. This is because it scales linearly as the number of threads increase.

Architecture

Processor architecture is the foundation of performance differences. A multi-core machine can run multiple CPU-intensive programs faster than a single core machine, but a single core system can execute a spin lock and read a variable faster. This is especially apparent if the multi-core system is reading shared data; the penalty imposed by the bus and cache system is significant compared to single core performance. Using a different architecture will change the systems bottlenecks.

MIMD Architecture

A main feature of multiple instruction multiple data architecture is the shared bus. Each core can access global memory at the cost of communicating with each other

Lock Contention

There are two basic lock contention strategies. The first is the “keep trying” approach. The current process will try and use as many CPU cycles as possible and the other processes will wait. This system relies on the operating system enforcing fair CPU usage and is more useful when delays are short.

The other approach is for processes to voluntarily “give up” the processor. This system is better if long delays exist because it has better cache performance. It is also the better choice for uniprocessor machines.

Test-and-Set

A test-and-set instruction writes a boolean value and returns the prior one. If the prior value is true, another process has the lock and you must wait. If it is false, the lock is free and may be acquired. The lock can also be released by writing false. Test-and-set instructions do not use a queue for waiting processes. As this code snippet shows,

```
class TASlock {

    AtomicBoolean state = new AtomicBoolean(false);


    void lock() {
```

```
                while (state.getAndSet(true)) {}

        }

        void unlock() {

                state.set(false);

}       }
```

all of the waiting processes can be inside the while loop simultaneously, so there is no need for a queue.

Space Complexity

A test-and-set lock has a relatively small footprint. It's read modify loop has a space complexity of O(1), which is better than the Peterson and Bakery's space of O(n).

Test-and-Set Performance

The ideal performance of test-and-set is constant. There is no speed up because of the sequential bottleneck. However, benchmarks show that test-and-set has an exponential runtime.

Test-and-Test-and-Set Locks

Test-and-test-and-set locks are an optimization of TAS instructions. They use a second while loop to stagger entry into the TAS loop. This is called the “lurking” stage, and the lock will spin until conditions occur that would make the lock free. As soon as the lock looks available it enters the “pouncing” stage, where the normal TAS instructions are executed.

```
class TTASlock {

        AtomicBoolean state =
 new AtomicBoolean(false);


        void lock() {

                while (true) {
// wait until lock “looks available”
                        while (state.get()) {}
 //normal TAS instructions
                                if (!state.getAndSet(true))

                                        return;
        }

}
```

While the test-and-test-and-set locks performs better than a TAS lock, it still has an exponential runtime and does not approach the ideal constant runtime. A more detailed model that performs as expected in field tests is needed. TTAS performs better because there is less synchronization that has to occur.

Compare and Swamp

Compare and Swap is an atomic CPU instruction that checks a value in a memory location, and if they are the same, writes a new value to that location. This single instruction is performed by hardware and the read write synchronization is not the bottleneck. Here is an implementation of CAS:

```
public class SimulatedCAS {

    private int value;

    public synchronized int getValue() { return value; }


public synchronized int compareAndSwap(int expectedValue, int newValue) {

        int oldValue = value;

        if (value == expectedValue)

            value = newValue;

        return oldValue;

    }
```

And this is a lock-free version:

```
public class CasCounter {

    private SimulatedCAS value;

    public int getValue() {
        return value.getValue();

    }

    public int increment() {

        int oldValue = value.getValue();

        while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)

            oldValue = value.getValue();

        return oldValue + 1;

    }
```

Lock free denotes that some thread will always make progress and starvation is permitted.

Bus-Based Architecture

A bus is a method of communication between processors. The cost of retrieving data from memory is an order of magnitude greater than going to cache; using a bus enables shared data to be retrieved from another processors cache and prevent this costly lookup. The bus checks other cache's for the requested data and copies it to the selected processor's cache. The bus is a very complicated piece of hardware, more so than the processor, and has many optimizations. For systems with more than 48 cores the bus becomes the primary bottleneck for increasing performance and efficiency.

A graphics processing unit does not have a bus, so all instructions must be coordinated through memory. Each core functions as independent unit and their instructions must all be planned out in advance.

Write-Back Cache

A write-back cache is a more efficient cache implementation. Each cache notifies every other cache of a write through the bus. This solution tracks which locations have been written to and marks them; this data is written to main memory when the data is flushed from the cache.