Mechanical Sympathy

Hardware and software working together in harmony

Saturday, 19 November 2011

Java Lock Implementations

We all use 3rd party libraries as a normal part of development. Generally, we have no control over their internals. The libraries provided with the JDK are a typical example. Many of these libraries employ locks to manage contention.

JDK locks come with two implementations. One uses atomic CAS style instructions to manage the claim process. CAS instructions tend to be the most expensive type of CPU instructions and on x86 have memory ordering semantics. Often locks are un-contended which gives rise to a possible optimisation whereby a lock can be biased to the un-contended thread using techniques to avoid the use of atomic instructions. This biasing allows a lock in theory to be quickly reacquired by the same thread. If the lock turns out to be contended by multiple threads the algorithm with revert from being biased and fall back to the standard approach using atomic instructions. Biased locking became the default lock implementation with Java 6.

When respecting the single writer principle, biased locking should be your friend. Lately, when using the sockets API, I decided to measure the lock costs and was surprised by the results. I found that my un-contended thread was incurring a bit more cost than I expected from the lock. I put together the following test to compare the cost of the current lock implementations available in Java 6.

The Test

For the test I shall increment a counter within a lock, and increase the number of contending threads on the lock. This test will be repeated for the 3 major lock implementations available to Java:

1. Atomic locking on Java language monitors
2. Biased locking on Java language monitors
3. `ReentrantLock` introduced with the java.util.concurrent package in Java 5.

I'll also run the tests on the 3 most recent generations of the Intel CPU. For each CPU I'll execute the tests up to the maximum number of concurrent threads the core count will support.

The tests are carried out with 64-bit Linux (Fedora Core 15) and Oracle JDK 1.6.0_29.

The Code

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.CyclicBarrier;

import static java.lang.System.out;

public final class TestLocks implements Runnable
{
    public enum LockType { JVM, JUC }
    public static LockType lockType;

    public static final long ITERATIONS = 500L * 1000L *1000L;
    public static long counter = 0L;

    public static final Object jvmLock = new Object();
    public static final Lock jucLock = new ReentrantLock();
    private static int numThreads;
    private static CyclicBarrier barrier;

    public static void main(final String[] args) throws Exception
    {
        lockType = LockType.valueOf(args[0]);
        numThreads = Integer.parseInt(args[1]);

        runTest(numThreads); // warm up
        counter = 0L;

        final long start = System.nanoTime();
        runTest(numThreads);
        final long duration = System.nanoTime() - start;

        out.printf("%d threads, duration %,d (ns)\n", numThreads, duration);
        out.printf("%,d ns/op\n", duration / ITERATIONS);
        out.printf("%,d ops/s\n", (ITERATIONS * 1000000000L) / duration);
        out.println("counter = " + counter);
```

Search This Blog

Search

Discussion Group

https://groups.google.com/forum/#!forum/mechanical-sympathy

About Me

Martin Thompson

London, United Kingdom

Technology geek exploring the capabilities of modern hardware. Available for development, training, performance tuning, and consulting services via Real Logic Limited. Twitter: @mjpt777

View my complete profile

Training & Consulting

www.real-logic.co.uk

Next public course

Conferences & Workshops

- QCon London March
- JavaLand Brühl - March
- Craft Conf Budapest - April
- JOnTheBeach Málaga - May
- Goto Amsterdam - June
- JCrete - July

Popular Posts

Java Garbage Collection Distilled

Serial, Parallel, Concurrent, CMS, G1, Young Gen, New Gen, Old Gen, Perm Gen, Eden, Tenured, Survivor Spaces, Safepoints, and the hundreds ...

CPU Cache Flushing Fallacy

Even from highly experienced technologists I often hear talk about how certain operations cause a CPU cache to "flush". This see...

Compact Off-Heap Structures/Tuples In Java

In my last post I detailed the implications of the access patterns your code takes to main memory. Since then I've had a lot of quest...

Memory Access Patterns Are Important

In high-performance computing it is often said that the cost of a cache-miss is the largest performance penalty for an algorithm. For many...

Native C/C++ Like Performance For Java Object Serialisation

Do you ever wish you could turn a Java object into a stream of bytes as fast as it can be done in

```
    }

    private static void runTest(final int numThreads) throws Exception
    {
        barrier = new CyclicBarrier(numThreads);
        Thread[] threads = new Thread[numThreads];

        for (int i = 0; i < threads.length; i++)
        {
            threads[i] = new Thread(new TestLocks());
        }

        for (Thread t : threads)
        {
            t.start();
        }

        for (Thread t : threads)
        {
            t.join();
        }
    }

    public void run()
    {
        try
        {
            barrier.await();
        }
        catch (Exception e)
        {
            // don't care
        }

        switch (lockType)
        {
            case JVM: jvmLockInc(); break;
            case JUC: jucLockInc(); break;
        }
    }

    private void jvmLockInc()
    {
        long count = ITERATIONS / numThreads;
        while (0 != count--)
        {
            synchronized (jvmLock)
            {
                ++counter;
            }
        }
    }

    private void jucLockInc()
    {
        long count = ITERATIONS / numThreads;
        while (0 != count--)
        {
            jucLock.lock();
            try
            {
                ++counter;
            }
            finally
            {
                jucLock.unlock();
            }
        }
    }
}
```

Script the tests:

```
set -x
for i in {1..8}; do java -XX:-UseBiasedLocking TestLocks JVM $i; done
for i in {1..8}; do java -XX:+UseBiasedLocking TestLocks JVM $i; done
for i in {1..8}; do java TestLocks JUC $i; done
```

The Results

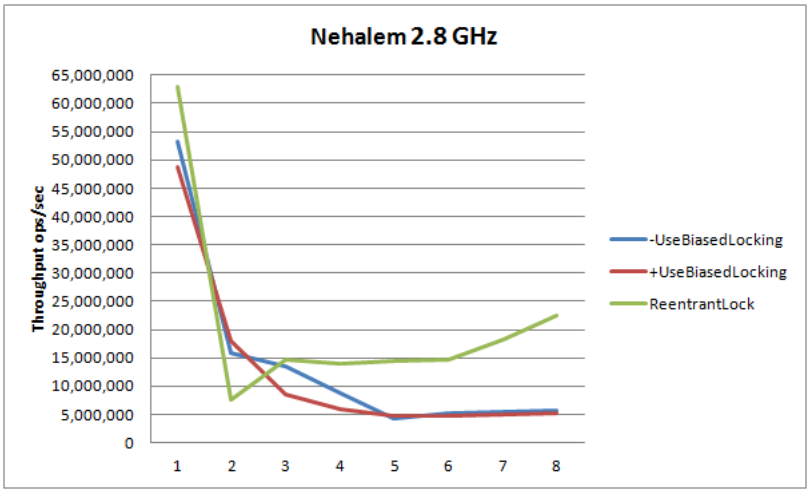

Figure 1.

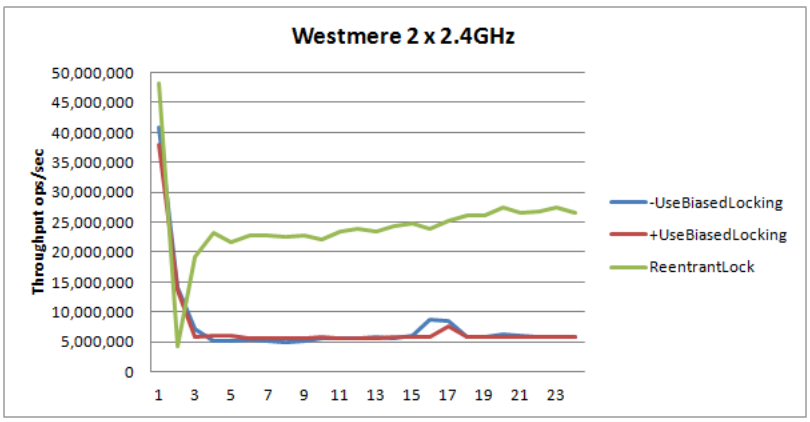

Figure 2.

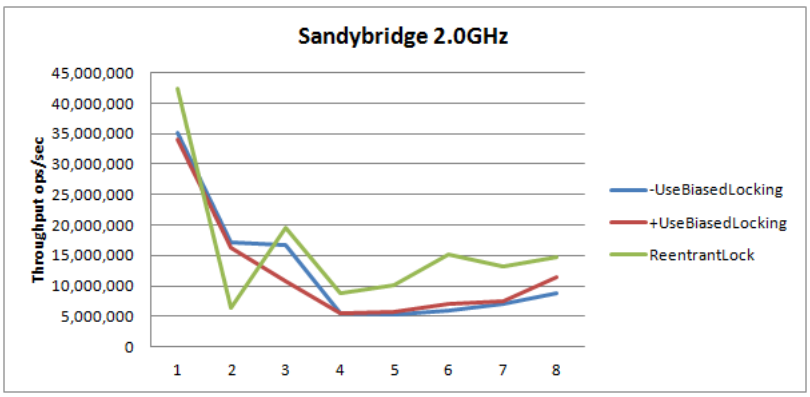

Figure 3.

Observations

1. Biased locking, in the un-contended case, is ~10% more expensive than the atomic locking. It seems that for recent CPU generations the cost of atomic instructions is less than the necessary housekeeping for biased locks. Previous to Nehalem, lock instructions would assert a lock on the memory bus to perform the these atomic operations, each would cost more than 100 cycles. Since Nehalem, atomic instructions can be handled local to a CPU core, and typically cost only 10-20 cycles if they do not need to wait on the store buffer to empty while enforcing memory ordering semantics.
2. As contention increases, language monitor locks quickly reach a throughput limit regardless of thread count.
3. ReentrantLock gives the best un-contended performance and scales significantly better with increasing contention compared to language monitors using synchronized.
4. ReentrantLock has an odd characteristic of reduced performance when 2 threads are contending. This deserves further investigation.
5. Sandybridge suffers from the increased latency of atomic instructions I detailed in a previous article when contended thread count is low. As contended thread count continues to increase, the cost of the kernel arbitration tends to dominate and Sandybridge shows its strength with increased memory throughput.

Conclusion

Biased locking should no longer be the default lock implementation on modern Intel processors. I recommend you measure your applications and experiement with the `-XX:-UseBiasedLocking` JVM option to determine if you can benefit from using atomic lock based algorithm for the un-contended case.

When developing your own concurrent libraries I would recommend `ReentrantLock` rather than using the synchronized keyword due to the significantly better performance on x86, if a lock-free alternative algorithm is not a viable option.

Update 20-Nov-2011

Dave Dice has pointed out that biased locking is not implemented for the locks created in the first few seconds of the JVM startup. I'll re-run my tests this week and post the results. I've had some more quality feedback that suggests my results could be potentially invalid. Micro benchmarks can be tricky but the advice of measuring your own application in the large still stands.

A re-run of the tests can be seen in this follow-on blog taking account of Dave's feedback.

Posted by Martin Thompson at 02:57

Labels: Java, Locks, Performance
Location: London, UK

5 comments:

Unknown 19 November 2011 at 15:11

Martin, it's indeed true that as CAS latency decreases (thanks to the efforts of modern processor designers), the utility of biased locking decreases, and that at some point in the near future it should be turned off.

Having said, that, biased locking will only provide benefit where there's no contention, in the single-threaded case in your benchmarks. Also, we disable biased locking on objects created during the early phase a of run, so it's having no benefit in your benchmark. You might try -XX:BiasedLockingStartupDelay=0. (This made a big difference at one thread in some quick tests I tried). Even better, you might institute multiple sub-runs (say 5, at about 10 seconds each) to give the JIT a chance to completely warm up.

Also, I should note that there are some changes in the pipeline that'll make synchronized faster than ReentrantLock for this type of benchmark.

Regards, -Dave http://blogs.sun.com/dave

Reply

Martin Thompson 19 November 2011 at 15:24

Thanks Dave,

I've had similar feedback from Gil Tene. I'll re-run my tests this week based on your suggestions and post findings.

It is the un-contended case I'm most interested in. In the low-latency trading space, contended locks impose such huge cost that they can render a system totally unsuitable.

I'd love to know what the plans are to make synchronized faster than ReentrantLock. You can see from my results how well it does in comparison under heavy contention.

Martin...

Reply

Martin Thompson 19 November 2011 at 15:37

Dave can you explain why biased locks are consistently more expensive even in the case of of my tests above if they do not kick in until the JVM has be running a few seconds?

Reply

Trevor Bernard 25 December 2011 at 04:36

What plotting library did you use to generate your graphs?

Reply

Martin Thompson 25 December 2011 at 10:34

Trevor,

I format the data in bash scripts and then load it into Excel for the graphs.

Reply

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.