Java theory and practice: Anatomy of a flawed microbenchmark

Is there any other kind?

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix

22 February 2005

Software engineers are notoriously obsessed, sometimes excessively, with performance. While sometimes performance is the most important requirement in a software project, as it might be when developing protocol routing software for a high-speed switch, most of the time performance needs to be balanced against other requirements, such as functionality, reliability, maintainability, extensibility, time to market, and other business and engineering considerations. In this month's *Java theory and practice*, columnist Brian Goetz explores why it is so much harder to measure the performance of Java language constructs than it looks.

View more content in this series

Even when performance is not a key requirement, or even a stated requirement, of the project you're working on, it is often difficult to ignore performance concerns, because you may think that would make you a "bad engineer." Toward the end of writing high-performance code, developers often write small benchmark programs to measure the relative performance of one approach against another. Unfortunately, as you learned in December's installment "Dynamic compilation and performance management", assessing the performance of a given idiom or construct in the Java language is far harder than it is in other, statically compiled languages.

A flawed microbenchmark

After publication of my October article, "More flexible, scalable locking in JDK 5.0," a colleague sent me the `SyncLockTest` benchmark (shown in Listing 1), which was purported to determine whether the `synchronized` primitive or the new `ReentrantLock` class was "faster." After running it on his laptop, he came to the conclusion that synchronization was faster, contrary to the conclusion of the article, and presented his benchmark as "evidence." The entire process -- the design of the microbenchmark, its implementation, its execution, and the interpretation of its results were flawed in a number ways. The colleague in this case is a pretty smart guy who's been around the block a few times, which goes to show how difficult this stuff can be.

Listing 1. Flawed SyncLockTest microbenchmark

```
interface Incrementer {
  void increment();
}

class LockIncrementer implements Incrementer {
  private long counter = 0;
  private Lock lock = new ReentrantLock();
  public void increment() {
    lock.lock();
    try {
      ++counter;
    } finally {
      lock.unlock();
    }
  }
}

class SyncIncrementer implements Incrementer {
  private long counter = 0;
  public synchronized void increment() {
    ++counter;
  }
}

class SyncLockTest {
  static long test(Incrementer incr) {
    long start = System.nanoTime();
    for(long i = 0; i < 10000000L; i++)
      incr.increment();
    return System.nanoTime() - start;
  }

  public static void main(String[] args) {
    long synchTime = test(new SyncIncrementer());
    long lockTime = test(new LockIncrementer());
    System.out.printf("synchronized: %1$10d\n", synchTime);
    System.out.printf("Lock:         %1$10d\n", lockTime);
    System.out.printf("Lock/synchronized = %1$.3f",
      (double)lockTime/(double)synchTime);
  }
}
```

`SyncLockTest` defines two implementations of an interface, and uses `System.nanoTime()` to time the execution of each implementation 10,000,000 times. Each implementation increments a counter in a thread-safe manner; one using built-in synchronization, and the other using the new `ReentrantLock` class. The stated goal was to answer the question, "Which is faster, synchronization or `ReentrantLock`?" Let's look at why this seemingly harmless-looking benchmark fails to measure what it purports to measure or, indeed, whether it measures anything useful at all.

Flaws in conception

Ignoring for the moment the implementation flaws, `SyncLockTest` also suffers from a more serious, conceptual flaw -- it misunderstands the question it is trying to answer. It purports to measure the performance costs of synchronization and ReentrantLock, techniques used to coordinate the action of multiple threads. However, the test program contains only a single thread, and therefore is guaranteed never to experience contention. It omits testing the very situation that makes locking relevant in the first place!

In early JVM implementations, uncontended synchronization was slow, a fact that was well-publicized. However, the performance of uncontended synchronization has improved substantially since then. (See Resources for a paper that describes some of the techniques used by JVMs to optimize the performance of uncontended synchronization.) On the other hand, contended synchronization was and still is far more expensive than uncontended synchronization. When a lock is contended, not only does the JVM have to maintain a queue of waiting threads, but it must use system calls to block and unblock the threads that are not able to acquire the lock immediately. Further, applications that experience a high degree of contention usually exhibit lower throughput, not only because more time is spent scheduling threads and less time doing actual work, but because CPUs may remain idle when threads are blocked waiting for a lock. A benchmark to measure the performance of a synchronization primitive should take into account a realistic degree of contention.

Flaws in methodology

This design failing was compounded by at least two failings of execution -- it was run only on a single-processor system (an unusual environment for a highly concurrent program, and whose synchronization performance may well differ substantially from multiprocessor systems), and only on a single platform. When testing the performance of a given primitive or idiom, especially one that interacts with the underlying hardware to such a significant degree, it is necessary to run benchmarks on many platforms before drawing a conclusion about its performance. When testing something as complicated as concurrency, a dozen different test systems, spanning multiple processors, and a range of processor counts (to say nothing of memory configurations and processor generations) would be advisable to start to get a picture of the overall performance of a given idiom.

Flaws in implementation

As to implementation, `SyncLockTest` ignores a number of aspects of dynamic compilation. As you saw in the December installment, the HotSpot JVM first executes a code path in interpreted mode, and only compiles it to machine code after a certain amount of execution. Failing to properly "warm up" the JVM can result in a significant skewing of performance measurements in two ways. First, the time taken by the JIT to analyze and compile the code path is included in the runtime of the test. More importantly, if compilation runs in the middle of your test run, your test result will be the sum of some amount of interpreted execution, plus the JIT compilation time, plus some amount of optimized execution, which doesn't give you much useful insight into the actual performance of your code. And if the code is not compiled prior to running your test, and not compiled during the test, then the entirety of your test run will be interpreted, which also doesn't give you much insight into the real-world performance of the idiom you are testing.

`SyncLockTest` also falls victim to the inlining and deoptimization problem discussed in December, where the first timing pass measures code that has been aggressively inlined with monomorphic call transformation, and the second pass measures code that has been subsequently deoptimized due to the JVM loading another class that extends the same base class or interface. When the timing test method is called with an instance of `SyncIncrementer`, the runtime will recognize that there is only one class loaded that implements `Incrementer`, and will convert virtual method calls to `increment()` into direct method calls to `SyncIncrementer`. When the timing test method is then

run with an instance of `LockIncrementer`, `test()` will be recompiled to use virtual method calls, meaning that the second pass through the `test()` harness method will be doing more work on each iteration than the first, turning our test into a comparison between apples and oranges. This can seriously distort the results, making whichever alternative that runs first look faster.

Benchmark code doesn't look like real code

The flaws discussed so far can be fixed with a reasonable amount of reworking of the benchmark code, introducing some test parameters such as the degree of contention, and running it on a greater variety of systems and for multiple values of the test parameters. But it also suffers from some flaws in approach, which may not be resolvable with any amount of tweaking. To see why this is so, you need to think like the JVM, and understand what is going to happen when `SyncLockTest` is compiled.

The Heisenbenchmark principle

When writing a microbenchmark to measure the performance of a language primitive like synchronization, you're grappling with the Heisenberg principle. You want to measure how fast operation X is, so you don't want to do anything else besides X. But often, the result is a do-nothing benchmark, which the compiler can optimize away partially or completely without you realizing it, making the test run faster than expected. If you put extraneous code Y into your benchmark, you're now measuring the performance of X+Y, introducing noise into your measurement of X, and worse, the presence of Y changes how the JIT will optimize X. Writing a good microbenchmark means finding that elusive balance between not enough filler and dataflow dependency to prevent the compiler from optimizing away your entire program, and so much filler that what you're trying to measure gets lost in the noise.

Because runtime compilation uses profiling data to guide its optimization, the JIT may well optimize the test code differently than it would real code. As with all benchmarks, there is a significant risk that the compiler will be able to optimize away the whole thing, because it will (correctly) realize that the benchmark code doesn't actually do anything or produce a result that is used for anything. Writing effective benchmarks requires that we "fool" the compiler into not pruning away code as dead, even though it really is. The use of the counter variables in the Incrementer classes is a failed attempt to fool the compiler, but compilers are often smarter than we give them credit for when it comes to eliminating dead code.

The problem is compounded by the fact that synchronization is a built-in language feature. JIT compilers are allowed to take some liberties with synchronized blocks to reduce their performance cost. There are cases when synchronization can be removed completely, and adjacent synchronization blocks that synchronize on the same monitor may be merged. If we're measuring the cost of synchronization, these optimizations really hurt us, as we don't know how many synchronizations (potentially all of them, in this case!) are being optimized away. Worse, the way the JIT optimizes the do-nothing code in `SyncTest.increment()` may be very different from how it optimizes a real-world program.

But it gets worse. The ostensible purpose of this microbenchmark was to test which was faster, synchronization or `ReentrantLock`. Because synchronization is built into the language, wheras

`ReentrantLock` is an ordinary Java class, the compiler will optimize a do-nothing synchronization differently than it will a do-nothing `ReentrantLock`-acquire. This optimization makes the do-nothing synchronization appear faster. Because the compiler will optimize the two cases both differently from each other and differently from the way it would in a real-world situation, the results of this program tell us very little about the difference between their relative performance in a real-world situation.

Dead code elimination

In December's article, I discussed the problem of dead-code elimination in benchmarks -- that because benchmarks often do nothing of value, the compiler can often eliminate whole chunks of the benchmark, distorting the timing measurements. This benchmark has that problem in several ways. The fact that the compiler can eliminate some code as dead is not necessarily fatal to our cause, but the problem here is that it will be able to perform different degrees of optimization on the two code paths, systematically distorting our measurements.

The two Incrementer classes purport to do some do-nothing work (incrementing a variable). But a smart JVM will observe that the counter variables are never accessed, and can therefore eliminate the code associated with incrementing them. And here is where we have a serious problem -- now the `synchronized` block in the `SyncIncrementer.increment()` method is empty, and the compiler may be able to entirely eliminate it, whereas `LockIncrementer.increment()` still contains locking code that the compiler may or may not be able to entirely eliminate. You might think that this code is a point in favor of synchronization -- that the compiler can more readily eliminate it -- but it's something that is far more likely to happen in do-nothing benchmarks than in real-world, well-written code.

It is this problem -- that the compiler can optimize one alternative better than the other, but the difference only becomes apparent in do-nothing benchmarks -- that makes this approach to comparing the performance of synchronization and `ReentrantLock` so difficult.

Loop unrolling and lock coalescing

Even if the compiler does not eliminate the counter management, it may still decide to optimize the two `increment()` methods differently. A standard optimization is loop unrolling; the compiler will unroll the loops to reduce the number of branches. How many iterations to unroll depends on how much code is inside the loop body, and there is "more" code inside the loop body of `LockIncrementer.increment()` than `SyncIncrementer.increment()`. Further, when `SyncIncrementer.increment()` is unrolled and the method call inlined, the unrolled loop will be a sequence of lock-increment-unlock groups. Because these all lock the same monitor, the compiler can perform lock coalescing (also called lock coarsening) to merge adjacent `synchronized` blocks, meaning that `SyncIncrementer` will be performing fewer synchronizations than might be expected. (And it gets worse; after coalescing the locks, the synchronized body will contain only a sequence of increments, which can be strength-reduced into a single addition. And, if this process is applied repeatedly, the entire loop can be collapsed into a single synchronized block with a single "counter=10000000" operation. And yes, real-world JVMs can perform these optimizations.)

Again, the problem is not strictly that the optimizer is optimizing away our benchmark, but that it is able to apply a different degree of optimization to one alternative than to another, and that the types of optimizations that it can apply to each alternative would not likely be applicable in real-world code.

Flaw scorecard

This list is not exhaustive, but here are just some of the reasons why this benchmark didn't do what its creator thought it did:

- No warmup was performed, and it made no account for the time taken by JIT execution.
- The test was susceptible to errors induced by monomorphic call transformation and subsequent deoptimization.
- The code protected by the synchronized block or `ReentrantLock` is effectively dead, which distorts how the JIT will optimize the code; it may be able to eliminate the entirety of the synchronization test.
- The test program wants to measure the performance of a locking primitive, but it did so without including the effects of contention, and was run only on a single-processor system.
- The test program was not run on a large enough variety of platforms.
- The compiler will be able to perform more optimization on the synchronization test than the `ReentrantLock` test, but not in a way that will help real-world programs that use synchronization.

Ask the wrong question, get the wrong answer

The scary thing about microbenchmarks is that they always produce a number, even if that number is meaningless. They measure something, we're just not sure what. Very often, they only measure the performance of the specific microbenchmark, and nothing more. But it is very easy to convince yourself that your benchmark measures the performance of a specific construct, and erroneously conclude something about the performance of that construct.

Even when you write an excellent benchmark, your results may be only valid on the system you ran it on. If you run your tests on a single-processor laptop system with a small amount of memory, you may not be able to conclude anything about the performance on a server system. The performance of low-level hardware concurrency primitives, like compare-and-swap, differ fairly substantially from one hardware architecture to another.

The reality is that trying to measure something like "synchronization performance" with a single number is just not possible. Synchronization performance varies with the JVM, processor, workload, JIT activity, number of processors, and the amount and character of code being executed using synchronization. The best you can do is run a series of benchmarks on a series of different platforms, and look for similarity in the results. Only then can you begin to conclude something about the performance of synchronization.

In benchmarks run as part of the JSR 166 (`java.util.concurrent`) testing process, the shape of the performance curves varied substantially between platforms. The cost of hardware constructs such as CAS varies from platform to platform and with number of processors (for example,

uniprocessor systems will never have a CAS fail). The memory barrier performance of a single Intel P4 with hyperthreading (two processor cores on one die) is faster than with two P4s, and both have different performance characteristics than Sparc. So the best you can do is try and build "typical" examples and run them on "typical" hardware, and hope that it yields some insight into the performance of our real programs on our real hardware. What constitutes a "typical" example? One whose mix of computing, IO, synchronization, and contention, as well as whose memory locality, allocation behavior, context switching, system calls, and inter-thread communication, approximates that of a real-world application. Which is to say that a realistic benchmark looks an awful lot like a real-world program.

How to write a perfect microbenchmark

So, how do you write a perfect microbenchmark? First, write a good optimizing JIT. Meet the people who have written other good, optimizing JITs (they're easy to find, because not too many good, optimizing JITs exist!). Have them over to dinner, and swap stories of performance tricks on how to run Java bytecode as fast as possible. Read the hundreds of papers on optimizing the execution of Java code, and write a few. You will then have the skills you need to write a good microbenchmark for something like the cost of synchronization, object pooling, or virtual method invocation.

Are you kidding?

You may think that the above recipe for writing a good microbenchmark is excessively conservative, but writing a good microbenchmark does require a great deal of knowledge of dynamic compilation, optimization, and JVM implementation techniques. To write a test program that will really test what you think it does, you have to understand what the compiler is going to do to it, the performance characteristics of the dynamically compiled code, and how the generated code differs from that of typical, real-world code that uses the same constructs. Without that degree of understanding, you won't be able to tell whether your program measures what you want it to.

So what do you do?

If you really want to know whether synchronization is faster than an alternate locking mechanism (or answer any similar micro-performance question), what do you do? One option (which won't sit well with most developers) is to "trust the experts." In the development of the `ReentrantLock` class, the JSR 166 EG members ran hundreds, if not thousands, of hours of performance tests on many different platforms, examined the machine code generated by the JIT, and pored over the results. Then they tweaked the code and did it again. A great deal of expertise and detailed understanding of JIT and microprocessor behavior went into the development and analysis of these classes, which unfortunately cannot be summarized in the results of a single benchmark program, as much as we would like them to be. The other option is to focus your attention towards "macro" benchmarks -- write some real-world programs, code them both ways, develop a realistic load-generation strategy, and measure the performance of your application using the two approaches under realistic load conditions and a realistic deployment configuration. That's a lot of work, but it will get you a lot closer to the answer you're looking for.

Resources

- Read the complete *Java theory and practice* series by Brian Goetz. December's column, Dynamic compilation and performance measurement, explores why it is that writing microbenchmarks for Java programs is so much harder than it is with statically compiled languages like C.
- Learn more about the new locking primitives in JDK 5.0 in More flexible, scalable locking in JDK 5.0 (developerWorks, October 2004).
- Want to know how synchronization is implemented? Thin Locks: Featherweight Synchronization for Java by David F. Bacon, Ravi Konoru, Chet Murthy, and Mauricio Serrano, shows some of the optimizations possible for reducing the cost of uncontended synchronization.
- To learn more about Java technology, visit the developerWorks Java zone. You'll find technical documentation, how-to articles, education, downloads, product information, and more.
- Visit the New to Java technology site for the latest resources to help you get started with Java programming.
- Get involved in the developerWorks community by participating in developerWorks blogs.
- Browse for books on these and other technical topics.

About the author

Brian Goetz

Brian Goetz has been a professional software developer for over 18 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. See Brian's published and upcoming articles in popular industry publications.