

Robust Java benchmarking, Part 1: Issues

Understand the pitfalls of benchmarking Java code

Skill Level: Advanced

[Brent Boyer \(ellipticgroupinc@gmail.com\)](mailto:ellipticgroupinc@gmail.com)

Programmer
Elliptic Group, Inc.

24 Jun 2008

Program performance is always a concern, even in this era of high-performance hardware. This article, the first in a two-part series, guides you around the many pitfalls associated with benchmarking Java™ code. [Part 2](#) covers the statistics of benchmarking and offers a framework for performing Java benchmarking. Because almost all new languages are virtual machine-based, the general principles the article describes have broad significance for the programming community at large.

Program performance, even in the age of multigigahertz/multicore processors and multigigabytes of RAM, remains a perennial concern. New, challenging applications (or increased programmer laziness) have matched every gain in hardware capability. Benchmarking code — and drawing correct conclusions from the results — has always been problematic, and few languages are trickier to benchmark than the Java language, especially on sophisticated modern virtual machines.

This two-part article addresses only program-execution time. It doesn't consider other important execution characteristics, such as memory usage. Even within this limited definition of performance, pitfalls abound in trying to benchmark code accurately. Their number and complexity make most attempts at "roll-your-own" benchmarking inaccurate and often misleading. This first part of the article is devoted to covering just these issues. It lays out the terrain you need to cover if you want to write your own benchmarking framework.

A performance puzzler

I'll start the discussion with a performance puzzler that illustrates some benchmarking issues. Consider the code in Listing 1 (see [Resources](#) for a link to the full sample code for this article):

Listing 1. Performance puzzler

```
protected static int global;

public static void main(String[] args) {
    long t1 = System.nanoTime();

    int value = 0;
    for (int i = 0; i < 100 * 1000 * 1000; i++) {
        value = calculate(value);
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-6) + " milliseconds");
}

protected static int calculate(int arg) {
    //L1: assert (arg >= 0) : "should be positive";
    //L2: if (arg < 0) throw new IllegalArgumentException("arg = " + arg + " < 0");

    global = arg * 6;
    global += 3;
    global /= 2;
    return arg + 2;
}
```

Which version runs fastest?:

- A. Leave the code as it is (no arg test inside calculate)
- B. Uncomment just line L1, but run with assertions disabled (use the `-disableassertions` JVM option; this is also the default behavior)
- C. Uncomment just line L1, but run with assertions enabled (use the `-enableassertions` JVM option)
- D. Uncomment just line L2

You should at least guess that A — having no test — must be fastest, with bonus points if you guess that B should be almost as fast as A, because with assertions off, line L1 is dead code that a good dynamic optimizing compiler should eliminate. Right? Unfortunately, you might be wrong. The code in [Listing 1](#) is adapted from Cliff Click's 2002 JavaOne talk (see [Resources](#)). His slides report these execution times:

- A. 5 seconds
- B. 0.2 seconds

- C. (He doesn't report this case)
- D. 5 seconds

The shock, of course, is B. How can it possibly be 25 times faster than A?

Six years later, I run the code in [Listing 1](#) on this modern configuration (which I use for every benchmark result in this article unless I note otherwise):

- Hardware: 2.2 GHz Intel Core 2 Duo E4500, 2 GB RAM
- Operating system: Windows® XP SP2 with all updates as of March 13, 2008
- JVM: 1.6.0_05, with `-server` used for all tests

I get:

- A. 38.601 ms
- B. 56.382 ms
- C. 38.502 ms
- D. 39.318 ms

B is now distinctly slower than A, C, and D. But the results are still strange: B ought to be the same as A, and the fact that it is slower than C is surprising. Note that I took four measurements for each configuration and obtained totally reproducible results (within 1 ms).

Click's slides discuss why he obtained his strange results. (They turn out to be due to complicated JVM behavior; also, a bug was involved.) Click is the architect of the HotSpot JVM, so it's no surprise that he came up with a rational explanation. But is there any hope that you, an ordinary programmer, can do correct benchmarks?

The answer is yes. In [Part 2](#) of this article, I present a Java benchmarking framework that you can download and use with confidence because it handles many of the benchmarking snares. The framework is easy to use for most benchmarking needs: just package the target code into some type of task object (either a `Callable` or `Runnable`) and then make a single call to the `Benchmark` class. Everything else — performance measurements, statistical calculations, and the result report — occurs automatically.

As a quick application of the framework, I'll rebenchmark the code in [Listing 1](#) by replacing `main` with the code in [Listing 2](#):

Listing 2. Performance puzzler solved using Benchmark

```
public static void main(String[] args) throws Exception {
    Runnable task = new Runnable() { public void run() {
        int value = 0;
        for (int i = 0; i < 100 * 1000 * 1000; i++) {
            value = calculate(value);
        }
    } };
    System.out.println("Cliff Click microbenchmark: " + new Benchmark(task));
}
```

Running the code on my configuration yields:

- A. mean = 20.241 ms ...
- B. mean = 20.246 ms ...
- C. mean = 26.928 ms ...
- D. mean = 26.863 ms ...

Finally, sanity: A and B have essentially the same execution time. And C and D (which do the same argument checking) also have about the same (slightly longer) execution time.

Using `Benchmark` yields the expected results in this case, probably because it internally executes `task` many times, with the "warmup" results discarded until the steady-state execution profile emerges, and then it takes a series of accurate measurements. In contrast, the code in [Listing 1](#) immediately starts measuring execution, which means that its results might have little to do with the actual code and more to do with [JVM behavior](#). Although I suppressed it in the results above (as indicated by the ...), `Benchmark` performs some powerful statistical calculations that tell you the results' reliability.

But don't just immediately use the framework. Familiarize yourself at some level with this whole article, particularly some of the tricky issues with [Dynamic optimization](#), as well as some of the interpretation problems I discuss in [Part 2](#). *Never blindly trust any numbers. Know how they were obtained.*

Execution-time measurement

In principle, measuring code-execution time is trivial:

1. Record the start time.

2. Execute the code.
3. Record the stop time.
4. Compute the time difference.

Most Java programmers probably instinctively write code similar to Listing 3:

Listing 3. Typical Java benchmarking code

```
long t1 = System.currentTimeMillis();
task.run(); // task is a Runnable which encapsulates the unit of work
long t2 = System.currentTimeMillis();
System.out.println("My task took " + (t2 - t1) + " milliseconds to execute.");
```

Listing 3's approach should usually be fine for long-running tasks. For example, if `task` takes one minute to execute, it's unlikely that the resolution issues I discuss below are significant. But as `task`'s execution time decreases, this code becomes increasingly inaccurate. A benchmarking framework should automatically handle any `task`, so Listing 3 warrants examination.

One problem is resolution: `System.currentTimeMillis`, as its name indicates, returns a result with only nominal millisecond resolution (see [Resources](#)). If you assume that its result includes a random ± 1 ms error, and you want no more than 1 percent error in the execution-time measurement, then `System.currentTimeMillis` fails for tasks that execute in 200 ms or less (because differential measurement involves two errors that could add up to 2 ms).

In reality, `System.currentTimeMillis` can have ~10-100 times worse resolution. Its Javadocs state:

Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds.

People have reported the figures in Table 1:

Table 1. Table using a heading tag

| Resolution | Platform | Source (see Resources) |
|------------|---------------------------------------|---|
| 55 ms | Windows 95/98 | Java Glossary |
| 10 ms | Windows NT, 2000, XP single processor | Java Glossary |
| 15.625 ms | Windows XP multi processor | Java Glossary |
| ~15 ms | Windows (presumably XP) | Simon Brown |

| | | |
|-------|------------------|---------------|
| 10 ms | Linux 2.4 kernel | Markus Kobler |
| 1 ms | Linux 2.6 kernel | Markus Kobler |

So, the code in [Listing 3](#) could easily start breaking down for tasks that execute in less than about 10 seconds.

A final issue with `System.currentTimeMillis` that affects even long-running tasks is that it is supposed to reflect "wall-clock" time. This means that its values can occasionally have abrupt leaps (backward or forward) in time that are due to events such as the change from standard time to daylight saving time, or Network Time Protocol (NTP) synchronization. These adjustments can, on rare occasions, cause erroneous benchmark results.

JDK 1.5 introduced a much higher-resolution API: `System.nanoTime` (see [Resources](#)). It nominally returns the number of nanoseconds since some arbitrary offset. Some of its key features are:

- It is useful only for differential time measurements.
- Its accuracy and precision (see [Resources](#)) should never be worse than (but may be as poor as) `System.currentTimeMillis`.
- On modern hardware and operating systems, it can deliver accuracy and precision in the microsecond range.

Conclusion: for benchmarking, always use `System.nanoTime`, because it usually has better resolution. But your benchmarking code must handle the possibility that it does no better than `System.currentTimeMillis`.

JDK 1.5 also introduced the `ThreadMXBean` interface (see [Resources](#)). It has several capabilities, but its `getCurrentThreadCpuTime` method has particular relevance for benchmarking (see [Resources](#)). This method offers the tantalizing possibility of measuring not the elapsed ("wall clock") time, but the actual CPU time used by the current thread, which is less than or equal to elapsed time.

Unfortunately, `getCurrentThreadCpuTime` has some problems:

- It might not be supported on your platform.
- Its semantics can differ across supported platforms. (For example, a thread that uses I/O might get billed the CPU time to do the I/O, or the time might be billed to an OS thread instead.)
- The `ThreadMXBean` Javadocs include this ominous warning: "Enabling thread CPU measurement could be expensive in some Java virtual machine implementations." (This is an OS-specific issue. On some OSs,

the microaccounting needed to measure thread CPU usage is always turned on, so `getCurrentThreadCpuTime` causes no additional performance hit. Others have it off by default; if enabled, it exhibits lower performance on all threads in the process or possibly all processes.)

- Its resolution is unclear. (Because it returns a result with nominal nanosecond resolution, it's natural to think that it has the same accuracy and precision limitations as `System.nanoTime`. However, I have not been able to find any documentation stating this, and one report states that it is much worse (see [Resources](#)). My experience with using `getCurrentThreadCpuTime` compared to `.nanoTime` is that it does tend to yield mean execution times that are smaller. On my desktop configuration, the execution times are about 0.5 to 1 percent smaller. Unfortunately, the measurement scatter is much higher; for example, the standard deviation could easily be three times larger. On an N2 Solaris 10 machine, execution times were 5 to 10 percent lower, and there was never an increase — sometimes there was a large decrease — in measurement scatter.)
- Worst of all: the CPU time used by the current thread can be irrelevant. Consider a task that has the calling thread (the current thread whose CPU time will be measured) merely establish a thread pool, then send a bunch of subtasks off to the pool, and then sit idle until the pool finishes. The CPU time used by the calling thread will be minimal, while the overall elapsed time to complete the task takes arbitrarily long. Thus, totally misleading execution times could be reported.

Because of these issues, it is too dangerous for a general-purpose benchmarking framework to use `getCurrentThreadCpuTime` by default. The `Benchmark` class presented in [Part 2](#) requires special configuration to enable it.

One word of caution about all of these time-measurement APIs: they have execution overhead, which affects how frequently they can be called before they overly distort the measurement. This effect is highly platform dependent. For example, on modern versions of Windows, `System.nanoTime` involves an OS call that executes in microseconds, so it should not be called more than once every 100 microseconds or so to keep the measurement impact under 1 percent. (In contrast, `System.currentTimeMillis` merely involves reading a global variable, so it executes extremely quickly, in nanoseconds. As far as measurement impact is concerned, it could be called more frequently, but because that global variable is not updated very often — about every 10 to 15 milliseconds according to [Table 1](#) — there's no point in calling it more frequently.) On the other hand, with most Solaris (and some Linux®) machines, `System.nanoTime` usually executes faster than `System.currentTimeMillis`.

Code warmup

In the [performance puzzler](#), I attributed Benchmark's sane results to the fact that it measures `task`'s steady-state execution profile, as opposed to the initial performance. Most Java implementations have a complicated performance life cycle. In general, the initial performance is usually relatively slow, and then it greatly improves for a while (usually in discrete leaps) until it reaches a steady state. Assuming that you want to measure this steady-state performance, you need to understand all the factors that lead up to it.

Class loading

JVMs typically load classes only when they're first used. So, a `task`'s first execution time includes the loading of all classes it uses (if they're not already loaded). Because class loading usually involves disk I/O, parsing, and verification, it can greatly inflate a `task`'s first execution. You can usually cure this effect by executing the `task` multiple times. (I say *usually* — instead of *always* — cured, because the `task` might have complicated branching behavior that causes it not to use all of its potential classes on any given execution. The hope is that if you execute the task enough times, these branches get fully explored and all relevant classes soon get loaded.)

If you use custom classloaders, another issue is that JVMs can decide to unload classes that have become garbage. This is likely not a major performance hit, but it is still less than ideal to have happen in the middle of your benchmark.

You can check whether or not class loading/unloading is occurring in the middle of your benchmark by calling the `getTotalLoadedClassCount` and `getUnloadedClassCount` methods of `ClassLoaderMXBean` before and after the benchmark (see [Resources](#)). If either result changed, then steady-state behavior has not been achieved.

Mixed mode

Modern JVMs typically let code run for a while (usually purely interpreted) in order to gather profiling information before doing Just-in-time (JIT) compilation (see [Resources](#)). What this means for benchmarking is that a task might need to execute many times before its steady-state execution profile emerges. For example, the current default behavior of Sun's client/server HotSpot JVM is that 1,500 (client) or 10,000 (server) calls must be made to a code block before the containing method is JIT compiled.

Note that I used the general phrase *code block*, which can refer not only to entire methods, but even to blocks within a method. For example, many JVMs are sophisticated enough to recognize that a block of code being looped over constitutes "hot" code, even if there's only a single call to the method that contains that block. I'll elaborate on this point in this article's [On-stack replacement](#) section.

So, benchmarking the steady-state performance requires something like:

1. Execute `task` once to load all classes.
2. Execute `task` enough times to ensure that its steady-state execution profile has emerged.
3. Execute `task` some more times to obtain an estimate of its execution time.
4. Use Step 3 to calculate n , the number of `task` executions whose cumulative execution time is sufficiently large.
5. Measure the overall execution time t of n more calls of `task`.
6. Estimate the execution time as t/n .

The goal behind measuring n executions of `task` ($n \geq 1$) is to make the cumulative execution time so large that all the time measurement errors I discuss above become insignificant.

Step 2 is tricky: how do you know when the JVM has finished optimizing the task?

You could try the seemingly clever approach of measuring execution times until they converge. This sounds good, but it fails if, say, the JVM was actually still profiling, and it suddenly applies that profiling to a JIT compile once you start Step 5; this could be especially problematic [in the future](#).

Furthermore, how do you quantify convergence?

Continuous compilation?

At present, Sun's HotSpot JVM merely does a single profiling phase followed by a possible compile. Ignoring [deoptimization](#), continuous compiling is currently not done because the overhead of the profiling code in hotspot methods is too severe (see [Resources](#)).

Solutions to this profiling-overhead problem are available. For instance, the JVM can retain two versions of methods: a fast one that contains no profiling code and a slow profiling one (see [Resources](#)). The JVM mostly uses the fast one but occasionally swaps in the slow one to maintain profiling information without heavily impacting performance. Or, perhaps the JVM concurrently executes the slow version whenever an otherwise idle core is available. Techniques like these might lead to continuous compilation being the norm in the future.

Another approach (which the `Benchmark` class uses) is simply to execute the task continuously for a predetermined, reasonably long time. A 10-second warmup phase should suffice (see page 33 of Click's talk). This approach might not be any more reliable than measuring the execution times until they converge, but it is simpler to

implement. It's also easier to parameterize: users should intuitively understand the concept and recognize that longer warmup times lead to more reliable results (at the cost of longer benchmarking times).

You can greatly increase your confidence about achieving steady-state performance if you can determine when JIT compilation occurs. In particular, if you think that you have achieved steady-state performance and start benchmarking, but then find that compilation occurred inside your benchmark, then you can abort and retry.

To my knowledge, no perfect way to detect JIT compilation exists. The best technique is to call `CompilationMXBean.getTotalCompilationTime` before and after a benchmark. Unfortunately, the implementation of `CompilationMXBean` was botched, so this approach has issues. Also note that another technique involves parsing (or manually watching) `stdout` when the `-XX:+PrintCompilation` JVM option is used (see [Resources](#)).

Dynamic optimization

Besides warmup issues, dynamic compilation done by JVMs involves several other concerns that affect benchmarking. They are subtle. Even worse, *the responsibility for coping with them lies solely with you, the benchmark programmer* — a benchmark framework can do little to address them. (This article's [Caching](#) and [Preparation](#) sections also discuss some issues that the benchmark programmer is responsible for, but those issues are mostly common sense.)

Deoptimization

One concern is deoptimization (see [Resources](#)): the JVM can stop using a compiled method and return to interpreting it for a while before recompiling it. This can happen when assumptions made by an optimizing dynamic compiler have become outdated. One example is class loading that invalidates monomorphic call transformations. Another example is *uncommon traps*: when a code block is initially compiled, only the most likely code path is compiled, while atypical branches (such as exception paths) are left interpreted. But if the uncommon traps turn out to be commonly executed, then they become hotspot paths that trigger recompilation.

So, even if you followed the advice in the preceding section and appear to have achieved steady-state performance, you need to be aware that performance could abruptly change. *This is one more reason why it is crucial to try to detect JIT compilation inside your benchmark.*

On-stack replacement

Another concern is on-stack replacement (OSR), an advanced JVM feature that helps optimize certain code structures (see [Resources](#)). Consider the code in Listing

4:

Listing 4. Example of code subject to OSR

```
private static final int[] array = new int[10 * 1000];
static {
    for (int i = 0; i < array.length; i++) {
        array[i] = i;
    }
}

public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) { // outer loop
        for (int j = 0; j < array.length; j++) { // inner loop 1
            result += array[j];
        }
        for (int j = 0; j < array.length; j++) { // inner loop 2
            result ^= array[j];
        }
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}
```

If the JVM solely kept count of method calls, then a compiled version of `main` would never be used because it is called only once. To solve this problem, JVMs can keep count of code-block executions inside of methods. In particular, with the code in Listing 4, the JVM can track how many times each loop is executed. (The end brace of a loop constitutes a "backward branch.") By default, any loop should trigger compilation of the entire method after 10,000 iterations or so. Because `main` is never called again, a simple JVM would never use this compiled code. However, a JVM using OSR is smart enough to replace the current code with the newer compiled code *in the middle of the method call*.

At first glance, OSR looks great. It seems as if the JVM can handle any code structure and still deliver optimum performance. Unfortunately, OSR suffers from a little-known defect: the code quality when OSR is used can be suboptimal. For instance, OSR sometimes cannot do loop-hoisting, array-bounds check elimination, or loop unrolling (see [Resources](#)). *If OSR is being used, you might not be benchmarking the top performance.*

Assuming that you want top performance, then the only cure for OSR is to recognize where it can occur and restructure your code to avoid it if possible. Typically this involves putting key inner loops in separate methods. For example, the code in [Listing 4](#) could be rewritten as shown in Listing 5:

Listing 5. Rewritten code no longer subject to OSR

```
public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) {    // sole loop
        result = add(result);
        result = xor(result);
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}

private static int add(int result) {    // method extraction of inner loop 1
    for (int j = 0; j < array.length; j++) {
        result += array[j];
    }
    return result;
}

private static int xor(int result) {    // method extraction of inner loop 2
    for (int j = 0; j < array.length; j++) {
        result ^= array[j];
    }
    return result;
}
```

In Listing 5, the `add` and `xor` methods will each be called 1,000,000 times, so they should get fully JIT compiled into optimal form. For this particular code, the first three runs measured execution times of 10.81, 10.79, and 10.80 seconds on my configuration. In contrast, the [Listing 4](#) code (which has all the loops inside `main` and therefore triggers OSR), has *twice* the execution time. (21.61, 21.61, and 21.6 seconds were its first three runs.)

One final comment about OSR: *it is usually only a performance problem in benchmarking*, when programmers are lazy and put everything in a single method such as `main`. In real applications, programmers naturally (we hope) write many finer-grained methods. Furthermore, code in which performance matters usually runs for a long time and invokes the critical methods many times. So, real-world code is usually not vulnerable to OSR performance problems. In your applications, don't be too anxious about it or mutilate otherwise elegant code over it (unless you can prove that it is an issue). Note that `Benchmark` by default executes the task several times in order to gather statistics, and these multiple executions have the nice side effect of eliminating OSR as a performance issue.

Dead-code elimination

The other subtle concern is dead-code elimination (DCE) (see [Resources](#)). In some circumstances, the compiler can determine that some code will never affect the output, and so the compiler will eliminate that code. Listing 6 shows the canonical example where this can be done statically (that is, at compile time, by `javac`):

Listing 6. Example of code subject to DCE

```
private static final boolean debug = false;

private void someMethod() {
    if (debug) {
        // do something...
    }
}
```

javac knows that the code inside the `if (debug)` block in Listing 6 will never get executed, and so it eliminates it. Dynamic compilers, especially once method inlining takes place, have many more ways to determine that code is dead. The problem with DCE during benchmarking is that the code that is executed can end up being only a small subset of your total code — entire computations might not even take place — which can lead to falsely short execution times.

I've been unable to find a good description of all the criteria that compilers can use to determine what constitutes dead code (see [Resources](#)). Unreachable code is obviously dead, but *JVMs often have more aggressive DCE policies*.

For example, reconsider the code in [Listing 4](#): note that `main` not only computes `result` but also uses `result` in the output that it prints. Suppose that I make just one tiny change and remove `result` from the `println`. In this case, an aggressive compiler might conclude that it does not need to compute `result` at all.

This is no mere theoretical concern. Consider the code in Listing 7:

Listing 7. Stopping DCE by using result in output

```
public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) { // sole loop
        result += sum();
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}

private static int sum() {
    int sum = 0;
    for (int j = 0; j < 10 * 1000; j++) {
        sum += j;
    }
    return sum;
}
```

I consistently find that the code in Listing 7 executes in 4.91 seconds on my configuration. If I modify the `println` statement to eliminate the reference to `result` — changing it to `System.out.println("Execution time: " +`

```
((t2 - t1) * 1e-9) + " seconds to compute result"); — I
```

consistently find that it executes in 0.08 seconds. Clearly DCE is eliminating the entire computation. (See [Resources](#) for another example of DCE.)

The only way to guarantee that DCE will not eliminate computations that you want to benchmark is to make the computations generate results, and then use the results somehow (for example, in output like the `println` in Listing 7). The `Benchmark` class supports this. If your task is a `Callable`, make sure that the computation is used to calculate the result returned by the `call()` method. If your task is a `Runnable`, make sure that the computation is used to calculate some internal state that is used by task's `toString` method (which must override the one from `Object`). If you obey these rules, `Benchmark` should completely prevent DCE.

Like OSR, DCE is usually not an issue for real applications (unless you are counting on code executing in a specific amount of time). Unlike OSR, however, DCE can be an enormous issue for poorly written benchmarks: OSR can merely lead to somewhat inaccurate results, whereas *DCE can lead to utterly wrong results*.

Resource reclamation

Typical JVMs automatically do two types of resource reclamation: garbage collection and object finalization (GC/OF). From the programmer's perspective, GC/OF is almost nondeterministic: it is ultimately outside of your control and can occur any time the JVM deems necessary.

In benchmarking, GC/OF times that are due to the task itself ought to be included in the result. For example, it is wrong to claim that a task is fast because its initial execution is short, if it eventually causes huge GC times. (But note that some tasks do not need to create objects. Instead, they just need to access already created objects. Consider a benchmark that aims to determine the time it takes to access an array element: the task should not create the array. Instead, the array should be created elsewhere, and its reference be made available to the task.)

But you also need to isolate the task's GC/OF from GC/OF caused by other code in the same JVM session. The only thing you can do is try to clean up the JVM before doing a benchmark, and also try to ensure that GC/OF that's due to the task itself is fully finished before the measurement ends.

The `System` class exposes the `gc` and `runFinalization` methods, which can be used for JVM cleanup. Beware that the Javadocs for these methods state only that "When control returns from the method call, the Java Virtual Machine has made a best effort to [do GC/OF]."

The `Benchmark` class I present in [Part 2](#) attempts to cope with GC/OF as follows:

1. Before doing any measurement, it calls a method named `cleanJvm`, which aggressively makes as many calls to `System.gc` and `System.runFinalization` as necessary until memory usage stabilizes and no objects remain to be finalized.
2. By default, it performs 60 execution measurements, each of which lasts at least 1 second (ensured by making multiple invocations of the task for each measurement if necessary). So the total execution time should be at least 1 minute, which should include enough GC/OF life cycles spread out over the 60 measurements that the full behavior is accurately sampled.
3. After all the measurements are over, it does one final call to `cleanJvm`, but this time it measures how long that takes. If this final cleanup is 1 percent or more of task's total execution time, then the benchmark report warns that GC/OF costs might not be truly accounted for in the measurements.
4. Because GC/OF acts like a noise source for each measurement, statistics are used to extract reliable conclusions.

A cautionary note: When I first wrote `Benchmark`, I tried to be clever and account for GC/OF costs inside each measurement using code like that shown in Listing 8:

Listing 8. Misleading way to account for GC/OF

```
protected long measure(long n) {
    cleanJvm();    // call here to cleanup before measurement starts

    long t1 = System.nanoTime();
    for (long i = 0; i < n; i++) {
        task.run();
    }
    cleanJvm();    // call here to ensure that task's GC/OF is fully included
    long t2 = System.nanoTime();
    return t2 - t1;
}
```

The problem is that calling `System.gc` and `System.runFinalization` inside the measurement loop can give a distorted view of the GC/OF cost. In particular, `System.gc` does a full garbage collection of all generations using a *stop-the-world* collector (see [Resources](#)). (That is the default behavior, but beware of JVM options such as `-XX:+ExplicitGCInvokesConcurrent` and `-XX:+DisableExplicitGC`.) In contrast, the garbage collector normally used by your application might operate quite differently. For example, it might be configured to work concurrently, and it might do many partial collections (especially of the young generation) with little effort. Likewise, finalizers are normally processed as a background task, so their cost is usually amortized across the system's idle time.

Caching

Hardware/operating system caches can sometimes complicate benchmarks. A simple example is file-system caching, which can take place in hardware or the OS. If you are benchmarking how long it takes to read the bytes from a file, but your benchmark code reads the same file many times (or you perform the same benchmark multiple times), then the I/O time can fall dramatically after the first read. If you want to benchmark random file reads, you likely need to ensure that different files are read to avoid caching.

CPU caching of main memory is so important that it deserves special attention (see [Resources](#)). For about 20 years now, CPUs have increased exponentially in speed, while main memory has weakly linearly increased in speed. To ameliorate this speed mismatch, modern CPUs use extensive caching (to the point where most of the transistors on a modern CPU are devoted to caching). A program that mates well with the CPU cache can have dramatically better performance than a program that doesn't. (Most real-world workloads achieve but a fraction of the CPU's theoretical throughput.)

Many factors affect how well a program mates with the CPU cache. For example, modern JVMs take great pains to optimize memory access: they might rearrange heap space, hoist values from the heap into the CPU register, do stack allocation, or perform object explosion (see [Resources](#)). But an important factor is simply the size of the data set. Let n characterize the size of the task's data set (for example, suppose it uses an array of length n). *Then any conclusions drawn from benchmarking with a single value of n can be highly misleading; you must do a series of benchmarks for various values of n .* An excellent example is in an article by J. P. Lewis and Ulrich Neumann (see [Resources](#)) They reproduce a graph of Java FFT performance relative to C as a function of n (the array size in this case) and find that Java performance oscillates between two times faster than C and two times slower, depending on which choice is made for n .

Preparation

Benchmarking pitfalls don't begin and end with the benchmarking framework you develop. You should also address several areas on your system before running any benchmark program on it.

Power

A low-level hardware problem, especially on laptops, is to make sure that power management (for example, Advanced Power Management [APM] or Advanced Configuration and Power Interface [ACPI]) does not make a state transition during the middle of your benchmark. Radical power-state changes, such as your computer

going into hibernation, probably will not result because of the CPU activity of the benchmark itself, or will be easily detected. Other power-state changes, however, are more insidious. Consider a benchmark that is initially CPU-bound, during which time the OS decides to power off the hard drive, and then the task wants to use the hard drive at the end of its run: the benchmark will finish, but the I/O portion may take longer. Another example includes systems that use Intel SpeedStep or similar technology to throttle CPU power dynamically. Before benchmarking, configure your OS to stop these effects.

Other programs

While benchmarking a task, you obviously should run no other programs (unless seeing how your task behaves on a loaded machine is the goal). And you likely want to shut down all nonessential background processes, as well as prevent scheduled processes (such as screen savers and virus scanners) from kicking in during benchmarking.

Windows offers the `ProcessIdleTask` API, which allows you to execute any pending idle processes before benchmarking. You can access this API by executing:

```
Rundll32.exe advapi32.dll,ProcessIdleTasks
```

from the command line. Be aware that it can take several minutes to execute, especially if you have not called it for a while. (Subsequent executions usually finish in several seconds.)

JVM options

Dozens of JVM options can affect benchmarking. Some relevant ones are:

- Type of JVM: server (`-server`) versus client (`-client`).
- Ensuring sufficient memory is available (`-Xmx`).
- Type of garbage collector used (advanced JVMs offer many tuning options, but be careful).
- Whether or not class garbage collection is allowed (`-Xnoclassgc`). The default is that class GC occurs; it has been argued that using `-Xnoclassgc` is a bad idea.
- Whether or not escape analysis is being performed (`-XX:+DoEscapeAnalysis`).
- Whether or not large page heaps are supported (`-XX:+UseLargePages`).
- If thread stack size has been changed (for example, `-Xss128k`).

- Whether or not JIT compiling is always used (`-Xcomp`), never used (`-Xint`), or only done on hotspots (`-Xmixed`; this is the default, and highest performance option).
- The amount of profiling that is accumulated before JIT compilation occurs (`-XX:CompileThreshold`), and/or background JIT compilation (`-Xbatch`), and/or tiered JIT compilation (`-XX:+TieredCompilation`).
- Whether or not biased locking is being performed (`-XX:+UseBiasedLocking`); note that JDK 1.6+ automatically does this.
- Whether or not the latest experimental performance tweaks have been activated (`-XX:+AggressiveOpts`).
- Enabling or disabling assertions (`-enableassertions` and `-enablesystemassertions`).
- Enabling or disabling strict native call checking (`-Xcheck:jni`).
- Enabling memory location optimizations for NUMA multi-CPU systems (`-XX:+UseNUMA`).

Conclusion to Part 1

Benchmarking is extremely difficult. Many factors, both obvious and subtle, can affect your results. To obtain accurate results, you need a thorough command of these issues, possibly by using a benchmarking framework that addresses some of them. Go to [Part 2](#) to learn about just such a robust Java benchmarking framework.

Resources

Learn

- [Companion site for this series](#): You'll find the full sample code and other supplementary material for this article series here.
- [How NOT To Write A Microbenchmark](#): See pages 11-22 of Cliff Click's JavaOne 2002 talk.
- [System class](#): API documentation for the `System` class, including its `currentTimeMillis`, `nanoTime`, `gc`, and `runFinalization` methods.
- [Java Glossary time entry](#) and [Millisecond accuracy in Java](#): Sources for the resolutions listed in Table 1.
- [Clocks and Timers — General Overview](#): A detailed discussion of `System.nanoTime`.
- [Accuracy and precision](#): Wikipedia explains these two terms.
- [ThreadMXBean](#): API documentation for the `ThreadMXBean` interface, including the `getCurrentThreadCpuTime` method.
- [Are you really Multi-Core?](#): Read about the particular relevance of `getCurrentThreadCpuTime` to benchmarking.
- [Accurate CPU timing on Windows — How?](#): This forum entry discusses issues with `getCurrentThreadCpuTime`.
- [ClassLoaderMXBean](#): API documentation for `ClassLoaderMXBean`, including its `getTotalLoadedClassCount` and `getUnloadedClassCount` methods.
- [Just-in-time compilation](#): Wikipedia's entry on JIT compilation.
- [Java Virtual Machine \(JVM\) — best way to tell when JIT compiling occurs?](#): A discussion of detecting JIT compilation.
- ["Java theory and practice: Dynamic compilation and performance measurement"](#) (Brian Goetz, developerWorks, December 2004): Read about deoptimization and other perils of benchmarking under dynamic compilation.
- [The Java HotSpot Performance Engine: On-Stack Replacement Example](#): OSR explained.
- [How to stop a compiler](#): Read about some limitations of OSR in the Dec 23, 2007 11:43:10 AM and Feb 25, 2008 8:31:53 AM follow-ups of this blog. The blog also includes an example of DCE.
- [Dead code elimination](#) and [Unreachable code](#): Wikipedia explains DCE and unreachable code.

- [Virtual Machine \(JVM\) - dead-code elimination — what are all the rules?:](#) According to this forum entry, understanding a compiler's criteria for identifying dead code is "really complex and changes with each new JVM version and so is impossible to articulate."
- [Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning:](#) Read about `System.gc`'s stop-the-world garbage collection.
- [CPU cache and Memory part 2: CPU caches:](#) Understand the importance of CPU caching.
- [High Performance Java Technology in a Multi-Core World:](#) How modern JVMs optimize memory access.
- [Performance of Java versus C++](#) (see the graph in the section "Don't characterize the speed of a language based on a single benchmark of a single program") and [CacheKiller — array size effects in cache performance:](#) Discussions of how data set size can affect performance.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone:](#) Find hundreds of articles about every aspect of Java programming.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Brent Boyer

Brent Boyer has been a professional software developer for more than nine years. He is the head of Elliptic Group, Inc., a software development company in New York, N.Y.

Trademarks

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.