Robust Java benchmarking, Supplement

Contents

Introduction

This document contains extra material that supplements the *Robust Java benchmarking* article.

Proper benchmarking, and associated areas like statistics, is sufficiently complicated, with so many details that matter, that it is impossible to write a comprehensive benchmarking article and still have it be accessible to the general developer. So, the material in the IBM developerWorks articles was a culling of the most essential and readable points. I owe a major debt to my technical editor, Brian Goetz, for his reader's perspective and feedback in this area. But there remains material for the individual seriously interested in benchmarking that needs to be documented somewhere. Hence this document.

In addition to the guidance of my technical editor, I would also like to thank Cliff Click, Doug Lea, David Holmes, J. P. Lewis, and Matthew Arnold for their reviews and feedback on early drafts of the article. Last, but not least, thanks to the staff at IBM developerWorks, especially Jennifer Aloi and my production editor Eileen Cohen.

On-stack replacement: details

The article's treatment of on-stack replacement (OSR) was a bit brief. It was mainly concerned with OSR's sometimes suboptimal code quality, and how if you structure your code poorly (as people are wont to do in benchmarks), you may draw wrong conclusions. However, there are a few more details worth mentioning here.

First, there are at least three situations where OSR is needed:

1) interpreted code → optimized native code. OSR is used to break out of long-running methods that are stuck in the interpreter. This is the case that was emphasized in the article.
2) optimized native code → interpreted code. This typically happens while debugging. Interpreted code may be required because the optimizing native code compiler eliminated the breakpoint.
3) speculative optimized native code → optimized native code. If the JVM performed speculative optimizations whose assumptions no longer hold, then OSR must be used to break out of otherwise invalid methods. The article mentioned this phenomena in its Deoptimization section, but should have pointed out that OSR is what is used to handle methods that would otherwise be stuck with invalid code.

Second, there are several ways that OSR affects performance:

a) the cpu cost to perform OSR. This is always negative.
b) it can both limit and enable certain optimizations. Negative: the mere possibility that OSR could occur, even if it is never performed, may force the JVM to keep information alive (e.g. not perform dead code elimination) because that information may be required if OSR must jump back to interpreted code. Positive: OSR allows aggressive speculative optimizations to be performed, which otherwise would have to be skipped if there was no way to undo them (i.e. do deoptimization).

An open question: exactly why does the code in Listing 4 run 2× slower than Listing 5? Is it because Listing 4's code had to run in interpreted mode a lot longer? Was it the CPU cost to do OSR? Or is it because of the inferior OSR code quality?

Alternatives to the mean

Part 2 of the article discussed the arithmetic mean as a measure of the central tendency of a population. It is what `Benchmark` reports. However, there are other centrality measures. I will mention a few here, and justify why `Benchmark` does not currently use them.

A common alternative to the mean is the median. It equals the mean for symmetric distributions, but differs for skewed distributions (although a theoretical bound is that it must remain within one standard deviation of the mean). It is sometimes well defined for fat-tailed PDFs like the Cauchy distribution where the mean is undefined. For performance measurements, it has the appealing property of being far less sensitive to outliers than the mean.

The median comes from a general family of nonlinear measures known as quantiles. For instance, the median is the same as the 2^{nd} quartile, 5^{th} decile, or 50^{th} percentile. Two other quantiles that have been proposed for performance measures are the 0^{th} and 100^{th} percentiles, more commonly known as the best (smallest) and worst (largest).

The argument for using the best (smallest execution time) is that since most noise sources should be positive (i.e. they increase the execution time), then taking the best should give the purest, most intrinsic result. But an argument can also be made for using the worst, because it allows you to make robust decisions, for instance, when quality of service matters.

Nevertheless, `Benchmark` sticks with the mean for several reasons. First, it is conventional: many people already understand the concept with no further explanation required. Second, the mean is sometimes favored over the median if the population is at least approximately Gaussian distributed because it is more efficiently estimated. Third (and probably related to its more efficient estimation), the mean is sometimes said to use all the information in the samples, unlike any quantile (e.g. best, median, worst). To see this, note that the mean changes whenever you change any sample value (i.e. if any sample changes by Δ, then the mean changes by Δ/n). In contrast, quantiles have a more complicated dependence on changes, but there are many scenarios in which the quantile does not change at all when a sample changes. For example, a quantile never changes its value when you change a sample that is initially larger then the quantile as long as the change keeps that sample larger than the original quantile. Fourth, the best and worst measurement times may be rejected since they are extremely sensitive to outliers. Finally, all of the quantile measures are nonlinear functions of the individual measurement values. This is a problem for `Benchmark`, because it appears to be impossible to extract the quantile statistics of the individual actions from the measured statistics,[1] whereas the mean and standard deviation are easily extractable.

It is a particular shame to not use the median. Its superior outlier insensitivity, relative to the mean, has already been mentioned. Furthermore, the scatter measure that is associated with it (average absolute deviation from the median) makes more sense than does the standard deviation/variance, which have a bizarre weighting whereby they *square* the deviation from the mean. This makes them even more sensitive to outliers than the mean, which uses equal weighting. (Are its convenient analytical properties the only reason why the standard deviation/variance is used?)

Mean and standard deviation estimators

Let x_i denote a series of n samples (e.g. measurements) of a random variable X (e.g. program execution time).

An estimator for the population's arithmetic mean using just these samples is

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

and an estimator for the standard deviation is

$$S_n = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2}$$

which are the formulas described in the text.

(1) is an unbiased estimator of the mean, which is good, but (2) is a biased estimator of the standard deviation, which is undesirable.

There exist other estimators for the standard deviation, the most famous of which is the so-called "sample standard deviation", which is given by

$$S_{n-1} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)^2} \tag{3}$$

S_{n-1}^2 is an unbiased estimator of the variance. However, taking the square root does not yield an unbiased estimator of the standard deviation. Instead, S_{n-1} has these three strikes against it: a) it too is a biased estimator, b) it has larger mean squared error (MSE) than S_n, and c) it is more complicated than S_n. Of these strikes, note that b) trumps all: *MSE is a better criteria than bias to use in choosing among estimators because it is more comprehensive* (MSE includes variance of the estimator along with its bias). *Thus, there is no reason why S_{n-1} should ever be favored over S_n.*[2]

For Gaussian distributed populations, there exists an unbiased estimator for the standard deviation that amounts to dividing S_{n-1} by a complicated constant. However, I could find no reference stating whether or not this estimator has lower MSE than S_n. Furthermore, it is preferable to minimize assumptions such as taking the PDF to be Gaussian.

Relating standard deviation to measurement scatter

If you do not wish to make any assumptions about the probability density function (PDF) except that it has finite variance, then the only general conclusions that can be drawn are given by Chebyshev's inequality, which says that the probability of a sample lying $\leq k$ standard deviations away from the mean is $\geq 1 - (1/k^2)$. Next, if the PDF is additionally assumed to be unimodal (almost certainly the case for benchmark measurements), then the Vysochanskiï-Petunin inequality gives even finer restrictions on the maximal spread. Finally, if you make even stronger assumptions about the PDF, such as that it belongs to some parametric family like the Gaussian, then even more powerful restrictions may apply. Table 1 summarizes this tradeoff between assumption strength and conclusion power:

Table 1. Relation between standard deviation and sample bounds

		probability sample within k standard deviations of mean		
		finite variance	**unimodal**	**Gaussian**
k	**1**	≥ 00.00%	≥ 55.56%	68.27%
	2	≥ 75.00%	≥ 88.89%	95.45%
	3	≥ 88.89%	≥ 95.06%	99.73%
	source	wikipedia	calculated	wikipedia

As you can see, for any PDF assumption, as k increases, you are guaranteed to include more of the possible values. Also, for any given value of k, as you make increasingly powerful assumptions, you increase the coverage.

Since the unimodal assumption should hold in benchmarking, while the Gaussian assumption may not, a reasonable conclusion from Table 1 is that at least 95% of all execution time measurements should lie within three standard deviations of the mean.

Block statistics versus action statistics

Definitions

The block statistics are those statistics (e.g. mean, standard deviation) *that are directly calculated from the measured execution times*.

I call them block statistics because, in general, each measurement involves a block of multiple task invocations. Recall from the Code warmup section of Part 1 that by default[3] `Benchmark` executes task enough times so that the cumulative execution time is large enough to diminish time measurement errors. If task takes a long time to execute, then $n = 1$ (only 1 execution is required to meet the time requirement), but if task executes in, say, 1 microsecond, then the cumulative result of $n = 10^6$ executions will be measured.

In addition to multiple task invocations, `Benchmark` allows the user to specify that the task internally executes multiple *identical* actions. For example, in Part 2 of the article where benchmarks of the data structure access times were carried out, it was useful to define an action as being a single data structure access. As can be seen from task code like Part 2's Listing 2, each task did multiple such accesses per invocation (via the loop inside `run`), which is why the parent code in Part 2's Listing 3 used the two argument version of the `Benchmark` constructor to specify the number of actions.

The action statistics are the statistics of these user defined actions. In general, they must be derived from the block statistics because they are not directly measured.

Calculation of the action statistics

As before, let n designate the number of task invocations in each measurement. Let m designate the number of actions per task invocation. Then each measurement is over $a = n \times m$ actions.

If $a = n = m = 1$, then the action statistics are the block statistics. However, in general, $a > 1$ so the action statistics must be calculated from the block statistics.

The mean of a random variable that is the sum of other random variables is always just the sum of the constituent means. Here, this means that the mean scales as a (i.e. divide the measurement's mean by a to get the action's mean).

The standard deviation is more complicated. The first step is knowing how the action execution times are interrelated. The simplest assumption is that they are independent identically distributed (iid). In this case, the standard deviation scales as $\sqrt{a}$ (i.e. divide the measurement's sd by $\sqrt{a}$ to get the action's sd). This is a corollary of the theorem that variances of uncorrelated random variables add. (Otherwise, if the action execution times are not iid but correlated, then the variance of their sum is the sum of their covariances. Since their covariances are, in general, unknowable, precise progress is impossible in this case.)

`Benchmark` operates as follows: the user tells it what m is (either explicitly as a constructor argument, or else the default value of 1 is assumed); it determines what n needs to be during the warmup phase; therefore it knows a. It does the measurements, calculates the block statistics directly from those measurements, and then calculates the action statistics from the block statistics using the scaling rules discussed above.

Result report

`Benchmark.toString` just reports the action statistics, since those are usually all that the user is interested in, while `Benchmark.toStringFull` reports those plus the block statistics that were actually measured.

Standard deviation measurement issues

This section argues that it can be difficult, perhaps impossible, to accurately measure the "true" standard deviation of the execution time for some tasks (e.g. microbenchmarks). This seems to be caused by environmental noise effects (e.g. operating system context switches). There appears to be no simple cure.

Lfsr: a task that should have minimal execution variation

Consider this `Runnable` task class:

Listing 1. Lfsr: a linear feedback shift register task class

```
protected static class Lfsr implements Runnable {

   protected int register = 1;   // stores the LFSR state
   protected int mask = 0xffffffff;
   protected int taps = (1 << 31) | (1 << 30) | (1 << 28) | (1 << 0);
   protected long numberTransitions = 1L * 1000L * 1000L;

   public void run() {
      for (long i = 0; i < numberTransitions; i++) {
         register = ((register >>> 1) ^ (-(register & 1) & taps)) & mask;
      }
   }

   @Override public String toString() { return String.valueOf(register); }   // part of DCE prevention

}
```

`Lfsr` implements a 32-bit linear feedback shift register (LFSR) and is a simplified version of a general class of LFSRs. It is defined as inner class inside `Benchmark` (see that version for more comments).

LFSRs are simple state machines. So, every time that you call `Lfsr.run`, the current state (stored in the `register` field) is advanced to the next state. `Lfsr.run` is essentially 2 lines of code: a loop head and its body. The loop body involves 6 bitwise and/or unary `int` operators, so it is very simple and should execute extremely fast. It is 100% CPU bound (no I/O, no synchronization). No objects are created by an `Lfsr` instance after it is constructed, in particular, *executing `run` repeatedly should never generate any garbage, so the garbage collector should never execute*. In spite of the simplicity of the computation, the LFSR state is pseudo-random, *so it should be impossible for a smart compiler to cut many corners and avoid doing the computations*.[4] The sum of all these properties is this: *once `run` has been fully JIT compiled, then it should execute with essentially no time variation* (assuming that the executing thread is continuously on the CPU, and nothing like CPU power throttling takes place). Think about it: what could be a source of execution time variation in the code above? Do the bitwise operations sometimes take a longer to execute? Of course not!

Variations actually seen in Lfsr

So what kinds of execution time variations do you actually see in `Lfsr`? Using `Benchmark` and calling its `toStringFull` method to get a complete report, I obtain this for the block statistics:

Listing 2. Bock statistics output for Listing 1

```
run #1: mean = 1.031 s (CI deltas: -394.666 us, +423.214 us), sd = 1.650 ms (CI deltas: -221.499 us,
    +368.772 us)
run #2: mean = 1.030 s (CI deltas: -260.489 us, +281.354 us), sd = 1.087 ms (CI deltas: -136.216 us,
    +206.500 us)
run #3: mean = 1.031 s (CI deltas: -247.205 us, +259.635 us), sd = 1.019 ms (CI deltas: -141.767 us,
    +188.889 us)
```

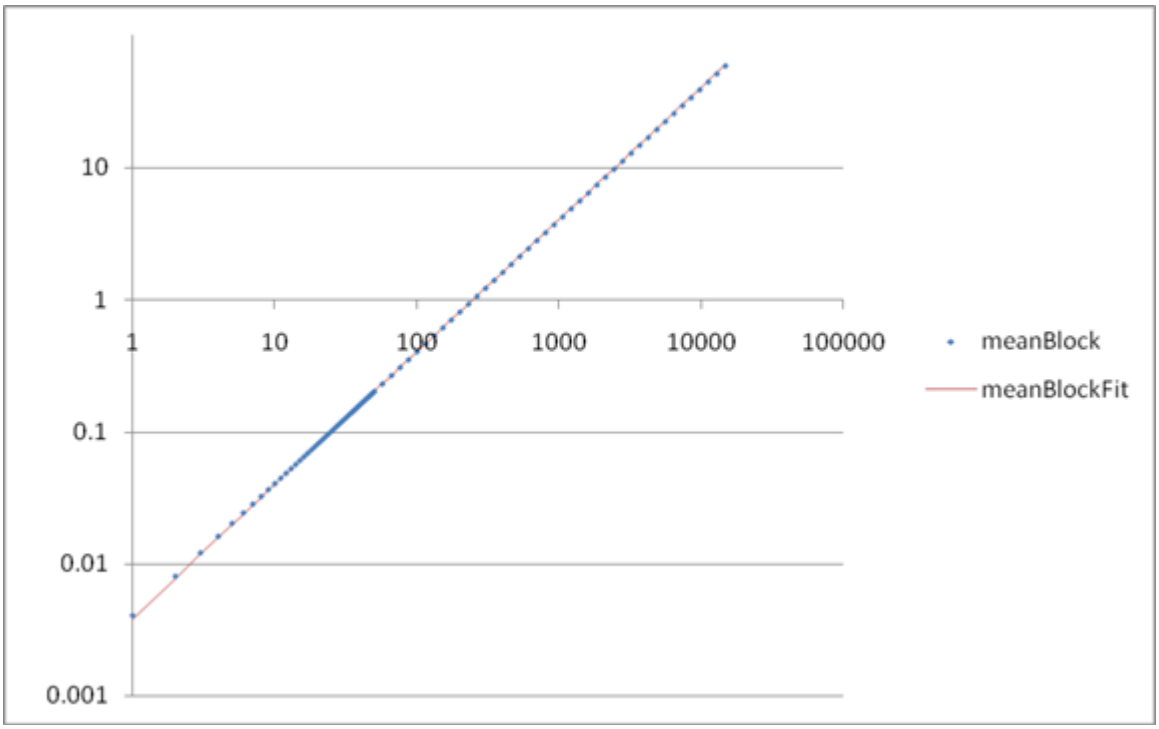
(Recall: the block statistics are obtained from the data that was actually measured. Ignore for now the action statistics, since they are calculated from the block statistics, and not directly measured; they will be discussed below.)

Cause of these variations

At first glance, the standard deviation (our measure for execution time variation) looks reasonable: 1-2 milliseconds is merely ~0.1-0.2% of the mean, and a few milliseconds of variation on a task that takes about a second to execute sounds like it might be due to time measurement error, say.

But this is not the case: time measurement error can be easily ruled out. Figure 1 is a graph of block execution time as a function of the number of times `Lfsr` is executed during the measurement:

Figure 1. Mean versus number of executions



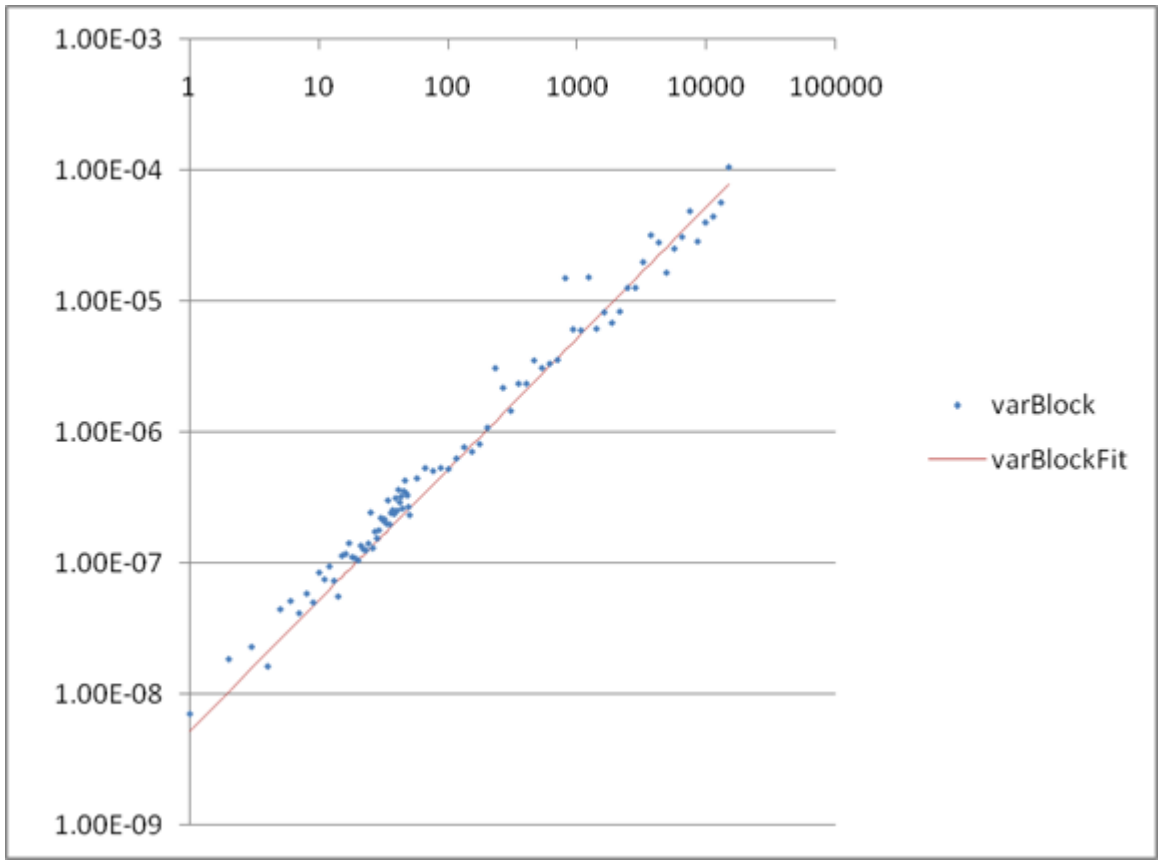
This graph is done using a log-log scale in order to clearly show the five decades of data. Along with the raw data, a fitted line is also shown. The almost perfect agreement of this line with the data proves that *data is essentially perfectly linear over a huge scale, even all the way down to 1 execution of* `Lfsr`*, which only takes a few milliseconds*. If time measurement error was on the order of a few milliseconds, different behavior would have been seen: the curve would stop decreasing linearly at small n, and instead saturate at some noise floor. Since this is not observed, *the 1-2 millisecond standard deviation above must come from another source besides measurement error*.

So what causes the observed execution time variations? I am pretty sure that the operating system is to blame. Specifically, that it is doing context switches of the benchmarking thread on a semi-random basis. Reasons:

1. Using the Microsoft performance counter,[5] I can confirm that my dual core XP machine is doing ~2-4,000 context switches per second. It is hard to imagine that the benchmarking thread is continuously using 1 of the cores.
2. My understanding of how Windows thread scheduling works is that a given thread has a time quantum (12 ms by default?) that it maximally runs for before it is guaranteed to be context switched from its core. It also could get switched before that quantum expires, but it is guaranteed to be switched by the end. (See the 27 Mar 2008, 1:48 PM UTC response by ioria74 in this forum.)
3. Other people attribute variation in timings to context switches

Regardless of cause, Figure 2 is a graph of the variance (square of the standard deviation) of the block execution time as a function of the number of times `Lfsr` is executed during the measurement:

Figure 2. Variance versus number of executions



Note how the fitted line, which is the simple equation $varBlockFit = cn$ with c as the only fit parameter (there is no constant fit parameter; the y-intercept is taken as 0), is an extremely good match. Now n iid random variables always have a variance that is n times the individual variance. Unfortunately, the converse is not true: just because the variance grows linearly versus n does not prove that it is made up of iid random variables. But it does suggest it, and my speculation that the iid noise sources are actually the context switch time variations is consistent with it.

Incidentally, the measurements reported above for my desktop were repeated on the N2 Solaris configuration mentioned in Part 2 of the article. The results are mostly qualitatively the same (e.g. superb linearity of block execution time as a function of the number of times `Lfsr` is executed) but the linearity of the variance is considerably less evident (in part, because the variance is so much smaller that it appears that noise is an issue).

More thoughts

So, `Lfsr` is a special task that should never have execution time variations, yet substantial variations are observed that are not due to time measurement error, but must be caused by something in the environment, such as context switches. Since this environmental disturbance should affect every process, therefore *it is impossible to measure the "true" standard deviation of a task unless its standard deviation is much larger than the environmental variations*.

So what tasks have sufficiently large "true" standard deviations that they rise above the environmental noise? *Probably any real application.* Said program will not be purely CPU bound, but will have lots of conditional logic/complex branching behavior, cache misses, multithreads and synchronization, I/O, garbage collection and object finalization, etc that should all act to cause variations in execution time above the environmental noise.

This leaves artificial microbenchmark-like programs, like `Lfsr` above, whose "true" standard deviation cannot be measured.

I am not aware of an easy way to reduce the environmental noise so that you can make more accurate measurements. I tried benchmarking `Lfsr` using these approaches:

1. Measuring CPU times instead of wall clock times. In theory this should work—for single threaded benchmarks, and if CPU times were perfectly measured—but in practice I have seen this cause *greater* measurement scatter.[6]
2. Boosting the main Java thread priority from normal to maximum, hoping that a high priority thread will be less frequently context switched. Result: no effect in environmental noise reduction.
3. Running Windows in Safe Mode, hoping that minimizing services might reduce the execution time variations. Result: if anything, the variations increased.
4. Setting the JVM process affinity to just one core of the CPU, hoping that maybe the main Java thread would continuously run on that core if the other core is set aside for background operating system processes. Result: if anything, the variations increased. This might be due to all the other JVM threads besides the main one now compteting for time on the same CPU.

One approach that I have not tried is using a real time operating system, and explicitly configuring it to run the benchmarking thread without any preemption. (Apparently, some operating systems have the concept of a "real time thread priority" which is a higher priority than even the normal operating system processes, which might achieve this effect). Also, Linux seems to support suppressing a CPU from responding to interrupts, which might also reduce context switches.

Standard deviation blowup

The excess execution time variation found above manifests itself in a related (and more radical) phenomena in the action statistics that I call "standard deviation blowup". Basically, you sometimes see the standard deviation become extremely large, even much larger than the mean, which seems unrealistic.

Examples of standard deviation blowup

You can already see the hint of this phenomena in `Lfsr`. Listing 2 shows the block statistics. Here is what the action statistics look like:

```
mean = 4.027 ms (CI deltas: -1.542 us, +1.653 us), sd = 103.149 us (CI deltas: -13.844 us, +23.048 us)
```

The sd/mean ratio for the block statistics is ~0.16%, but the corresponding quantity for the action statistics is ~2.56%, which is 16 times larger.

Here is a better example of "standard deviation blowup", obtained from a "nanobenchmark" (each task invocation does even less work than `Lfsr.run` does):

Listing 3. Code for a "nanobenchmark"

```
final StringBuilder sb = new StringBuilder("Yes");
Callable<StringBuilder> task = new Callable<StringBuilder>() {
   public StringBuilder call() { return sb.length() == 3 ? sb.replace(0, 3, "No") :
sb.replace(0, 2, "Yes"); }
};
System.out.println("StringBuilder.replace: " + new Benchmark(task).toStringFull());
```

Here, task boils down to doing a little `char[]` manipulation, so it executes extremely fast. No new objects are created over repeated invocations of `call`. I get this result (extraneous information removed):

```
action statistics: mean = 20.437 ns (CI deltas: -5.229 ps, +5.084 ps), sd = 168.910 ns (CI deltas: -22.590
ns, +32.759 ns)
   --then the number of actions per block measurement is a = 67108864
   --block statistics: mean = 1.371 s (CI deltas: -350.914 us, +341.209 us), sd = 1.384 ms (CI deltas: -
   185.055 us, +268.364 us)
```

So, each time measurement was over a block of 67,108,864 actions. Those block statistics look normal: it takes about 1.37 seconds to execute, and has a standard deviation of about 1.38 milliseconds, which is just 0.1% of the mean. However, the standard deviation of the action statistics is about 800% of the mean!

Cause of standard deviation blowup

Why did the action standard deviation blowup like this? Two reasons. First, I have already established above that the 1.3 ms standard deviation of the block statistics, although it seems small, is actually wrong: it is probably not from the task itself but comes from some environmental disturbance like the operating system context switching. The task's true standard deviation—whatever it is—is smaller than this. Second, the scaling rules differ between how the mean for the actions is calculated versus how the standard deviation is calculated: the mean scales by a, which is 67,108,864 for the benchmark above, while the standard deviation scales as $\sqrt{a}$, which is just 8,192 for the benchmark above. So, *miniscule noise in the block statistics gets magnified to dominating noise in the action statistics*.

Standard deviation warnings

Summary of the two preceding sections: the "true" standard deviation is sometimes impossible to measure, and this can manifest itself in standard deviation values for the action statistics that seem grossly inflated.

There seems to be no simple cure for this phenomena, but is there anything else that can be done? Yes: detect if it is happening, and warn the user. `Benchmark` currently takes two approaches.

First, `Benchmark` executes `Lfsr` under the exact same conditions (`Benchmark.Params` inner class) as the user supplied task in order to determine the environmental noise floor (which is taken as `Lfsr`'s standard deviation). By default, if this noise floor is 1% or more of task's standard deviation,[7] then task's standard deviation is considered to be affected.

This first approach is simple and easy to understand, but has one defect: the noise floor results are not always repeatable. Above, I showed the results of three runs in which the standard deviation varied by a factor of about 1.7, but, in fact, I have seen it vary by a factor of 3 or more.

To ensure that inaccurate standard deviations are detected, `Benchmark` uses a second approach that is based on a simple mathematical model which is applied to the action statistics. This model can detect situations in that the observed mean and standard deviation can only be explained by a small minority of outlier values.

If either approach finds something, then a warning is issued in the results report.

Standard deviation outlier model

Definitions

Let μ_B and σ_B^2 denote the mean and variance of the execution time measurements. It is irrelevant for the argument below whether these are the true population values, or merely sample estimates. But they are assumed to be known quantities, and are what I have been calling the block statistics, which is why the subscript B is used.

Let a denote the number of actions per measurement as before.

Let t_i where $i \in [1, a]$ designate the execution times of the actions that constitute some *hypothetical* measurement. Since the t_i are execution times, they cannot be negative. In fact, there is probably some positive number t_{min} that they realistically cannot be smaller than, so they satisfy

$$t_i \geq t_{min} > 0 \tag{4}$$

Let μ_A and σ_A^2 denote the mean and variance of the t_i. They correspond to what I have been calling the action statistics. By definition they obey

$$\mu_A = \frac{1}{a}\sum_{i=1}^{a} t_i > 0 \tag{5}$$

$$\sigma_A^2 = \frac{1}{a}\sum_{i=1}^{a} (t_i - \mu_A)^2 > 0 \tag{6}$$

Let a *scenario consistent with the measurements* be defined as some t_i that obey these equations:

$$\mu_B = a\mu_A = \sum_{i=1}^{a} t_i > 0 \tag{7}$$

$$\sigma_B^2 = a\sigma_A^2 = \sum_{i=1}^{a} (t_i - \mu_A)^2 > 0 \tag{8}$$

These equations follow naturally from assuming that the a actions—which are supposed to be identical actions—are iid. This implies that the mean and variance of a measurement that is the sum of these actions is therefore the sum of the actions' means and variances.

Note: there is not necessarily any specific measurement that was done whose action execution times should be identified with these t_i. Indeed, it is highly unlikely that any one of the measurements performed to determine μ_B and σ_B^2 had a series of t_i that satisfy those equations. Instead, the t_i are merely a series of action execution times that are plausible for a measurement. This is why I refer to it as a *hypothetical* measurement.

The simple outlier model proposed here is that the t_i are distributed as follows:

$$t_i = \begin{cases} a\ constant\ U, if\ i \leq c \\ a\ random\ pick\ from\ a\ Gaussian, if\ i > c \end{cases} \tag{9}$$

Here, I model every outlier as having the same constant value U where

$$U \gg \mu_A > 0 \tag{10}$$

is necessary in order to be considered a positive outlier. The quantity c is an integer that satisfies

$$1 \leq c \leq a \tag{11}$$

The non-outlier action execution times are modeled as coming from a Gaussian distribution with mean μ_g and variance σ_g^2. To be consistent with previous constraints like (4) and (10), require

$$\mu_A > \mu_g > t_{min} \tag{12}$$

$$\sigma_A > \sigma_g > 0 \tag{13}$$

$$\mu_g - t_{min} \gg \sigma_g \tag{14}$$

To be sure, (4) can never be perfectly obeyed because the left tail of the Gaussian always extends below t_{min}, but this error can easily be made as small as desired by making σ_g as small as needed. For instance, $\sigma_g < (\mu_g - t_{min})/4$ ensures that over 99.99% of the non-outlier action execution times are $> t_{min}$.

Note: I chose to put all the outliers in the lower subscripts and all the non-outliers in the upper subscripts. In reality, you would expect the outliers to be scattered among the non-outliers. As far as relating the observed (block) statistics to the unobserved (action) statistics is concerned, this order does not matter. I simply chose that order to make it easier to understand the math below.

Overview

Before plunging into the math and getting bogged down by details, an overview of the argument that follows is helpful.

The essence is that there can be certain values of a, μ_B, and σ_B^2 such that the model above limits the maximum valid value of c to something $< a$. Define this limit as c_{max}; typically $c_{max} \ll a$. In other words, I will show that *there are scenarios where the observed statistics are only explainable by a small number of outliers.*

Two features drive this restriction on valid values of c.

The first is that execution times must be positive (4). This means that most of the t_i have to be clustered near μ_A. There cannot be too many t_i that are $\gg \mu_A$ because no negative t_i are allowed to cancel them out and leave (5) satisfied.

The second is the linear nature of the mean requirement (7) versus the quadratic nature of the variance requirement (8). With the mean requirement, it does not matter how the $t_i$are distributed; each t_i makes an equal contribution in this sense: adding Δ to any of the t_i changes the mean by the same amount. In contrast, with the variance requirement, the larger t_i make more of a contribution to the variance than do the smaller values: adding Δ to one of the larger t_i changes the variance more than adding Δ to one of the smaller values. So *if a large variance requirement needs to be met, it pays to have a few really large values (i.e. outliers)* rather than spread the variance around more evenly.

Solution

Now let's do the math. Plugging (9) into (7) and (8) and then taking averages yields:

$$\mu_B = cU + (a-c)\langle t\rangle_{gauss} = cU + (a-c)\mu_g \tag{15}$$

$$\sigma_B^2 = c(U - \mu_A)^2 + (a-c)\langle (t-\mu_A)^2\rangle_{gauss} \tag{16}$$

Solving that last term takes a slight bit of work:

$$\begin{aligned}\langle (t-\mu_A)^2\rangle_{gauss} &= \int_{-\infty}^{\infty} dt PDF_{gauss}\ (t-\mu_A)^2 \\ &= \int_{-\infty}^{\infty} dt PDF_{gauss}\ \Big((t-\mu_g) - (\mu_A - \mu_g)\Big)^2 \\ &= \int_{-\infty}^{\infty} dt PDF_{gauss}\ \Big((t-\mu_g)^2 - 2(t-\mu_g)(\mu_A-\mu_g) + (\mu_A-\mu_g)^2\Big)\end{aligned} \tag{17}$$

Note that the first term is the variance, the middle term integrates to $-2(\mu_g - \mu_g)(\mu_A - \mu_g) = 0$, and the last term is constant. Then

$$\langle (t-\mu_A)^2\rangle_{gauss} = \sigma_g^2 + (\mu_A - \mu_g)^2 \tag{18}$$

Therefore (16) becomes

$$\sigma_B^2 = c(U - \mu_A)^2 + (a-c)\left(\sigma_g^2 + (\mu_A - \mu_g)^2\right) \tag{19}$$

Recap: a, μ_B, σ_B^2, μ_A, and σ_A^2 are known quantities. A functional form for the t_i has been assumed, but this model has 4 free parameters: c, U, μ_g, and σ_g^2. There are seven equations that must be obeyed: (10), (11), (12), (13), (14), (15), and (19). I solve these equations as follows: use (15) and (19) to solve for U and μ_g as a function of c and σ_g^2, and then see what values of c and σ_g^2 satisfy (10), (11), (12), (13), and (14). Ultimately, I will end up picking a value for σ_g^2.

Solving (15) and (19) for U and μ_g is easier if the variables are changed to

$$\widehat{U} \equiv U - \mu_A \tag{20}$$

$$\hat{\mu}_g \equiv \mu_g - \mu_A \tag{21}$$

Then (15) becomes

$$\begin{aligned}\mu_B = c(\widehat{U} + \mu_A) + (a-c)(\hat{\mu}_g + \mu_A) = c\widehat{U} + (a-c)\hat{\mu}_g + a\mu_A \\ \Rightarrow 0 = c\widehat{U} + (a-c)\hat{\mu}_g \\ \Rightarrow \widehat{U} = -\left(\frac{a-c}{c}\right)\hat{\mu}_g\end{aligned} \tag{22}$$

and (19) becomes

$$\sigma_B^2 = c\widehat{U}^2 + (a-c)(\sigma_g^2 + \hat{\mu}_g^2) \tag{23}$$

Define

$$\sigma_{Bg}^2 \equiv \sigma_B^2 - (a-c)\sigma_g^2 \tag{24}$$

then (23) becomes

$$\sigma_{Bg}^2 = c\widehat{U}^2 + (a-c)\hat{\mu}_g^2 \tag{25}$$

Substituting (22) into (25) yields

$$\sigma_{Bg}^2 = \frac{(a-c)^2}{c}\hat{\mu}_g^2 + (a-c)\hat{\mu}_g^2 = \left(\frac{a-c}{c}\right)(a-c+c)\hat{\mu}_g^2 = a\left(\frac{a-c}{c}\right)\hat{\mu}_g^2$$

$$\begin{aligned} \Rightarrow \hat{\mu}_g &= -\sqrt{\frac{c}{a(a-c)}}\,\sigma_{Bg} \\ \Rightarrow \mu_g &= \mu_A - \sqrt{\frac{c}{a(a-c)}}\,\sigma_{Bg} \\ &\approx \mu_A - \frac{\sqrt{c}}{a}\sigma_{Bg} \qquad if\ c \ll a \end{aligned} \tag{26}$$

Note: took the minus sign for $\hat{\mu}_g$ in (26) to satisfy the first inequality of (12).

Then (22) becomes

$$\begin{aligned} \widehat{U} &= \sqrt{\frac{a-c}{ac}}\,\sigma_{Bg} \\ \Rightarrow U &= \mu_A + \sqrt{\frac{a-c}{ac}}\,\sigma_{Bg} \\ &\approx \mu_A + \frac{1}{\sqrt{c}}\sigma_{Bg} \qquad if\ c \ll a \end{aligned} \tag{27}$$

So, (26) and (27) are solutions for U and μ_g as a function of c and σ_g^2.

The first inequality of both (10) and (12) is always satisfied given (26) and (27) if

$$\sigma_{Bg}^2 > 0 \tag{28}$$

which happens if

$$\sigma_g < \frac{1}{\sqrt{a-c}}\sigma_B \tag{29}$$

Since $\sigma_A = \frac{1}{\sqrt{a}}\sigma_B < \frac{1}{\sqrt{a-c}}\sigma_B$, one way to over satisfy (29) would be to insist that $\sigma_g < \sigma_A$, which is exactly what (13) asserts.

The second inequality of (10) is always satisfied, but the second inequality of (12) yields an equation that shows one way how c_{max} arises. In terms of $\hat{\mu}_g$, this second inequality is

$$\begin{aligned} 0 > \hat{\mu}_g &> t_{min} - \mu_A \\ \Rightarrow \hat{\mu}_g^2 &< (\mu_A - t_{min})^2 \\ \Rightarrow \frac{c}{a(a-c)}\sigma_{Bg}^2 &< (\mu_A - t_{min})^2 \\ \Rightarrow c\left(\sigma_B^2 - (a-c)\sigma_g^2\right) &< a(a-c)(\mu_A - t_{min})^2 \\ \Rightarrow \sigma_B^2 c - a\sigma_g^2 c + \sigma_g^2 c^2 &< a^2(\mu_A - t_{min})^2 - a(\mu_A - t_{min})^2 c \\ \Rightarrow \left(\sigma_g^2\right)c^2 + \left(\sigma_B^2 - a\sigma_g^2 + a(\mu_A - t_{min})^2\right)c &< a^2(\mu_A - t_{min})^2 \end{aligned} \tag{30}$$

Note that each coefficient of c on the left hand side is positive; in particular,

$$\sigma_B^2 - a\sigma_g^2 > 0 \tag{31}$$

by (8) and (13). *Therefore, the left hand side uniformly increases as c increases from 0.* But the right hand side is constant. Thus, in order for that inequality to hold, a maximum legitimate value of c exists. Define the upper bound imposed by this inequality as $c_{max\,1}$.

Solving for $c_{max\,1}$ is trivial because (30) is a quadratic equation in c. In particular, let

$$k_2 \equiv \sigma_g^2 > 0 \tag{32}$$

$$k_1 \equiv \sigma_B^2 - a\sigma_g^2 + a(\mu_A - t_{min})^2 > 0 \tag{33}$$

$$k_0 \equiv -a^2(\mu_A - t_{min})^2 < 0 \tag{34}$$

be the coefficients of c in (30) after the right hand side is brought over to the left. Because of the nature of the coefficients, it is obvious that the solutions are both real and distinct, one is positive and the other is negative, and that you should take the positive one. Then a robust floating point algorithm for the solution is

$$term = -\frac{1}{2}\left(k_1 + sgn(k_1)\sqrt{k_1^2 - 4k_2k_0}\right) \tag{35}$$

$$root_1 = term/k_2 \tag{36}$$

$$root_2 = k_0/term \tag{37}$$

where $c_{max\,1}$ is assigned to the positive root. Because $k_1 > 0$, the positive root will be $root_2$, so you can directly write

$$c_{max\,1} = \frac{-2k_0}{k_1 + \sqrt{k_1^2 - 4k_2k_0}} \tag{38}$$

Next is a tricky issue: what value should be chosen for σ_g? *Since you have no real idea what σ_g is, ultimately you have to make a guess.* One choice that seemingly satisfies (13) and (14) is:

$$\sigma_g = \min\left(\frac{\mu_g - t_{min}}{4}, \sigma_A\right) \tag{39}$$

The problem with this is that μ_g is not a constant—it is a function of c as given by (26).

There may be multiple ways to proceed here, but a simple approach is to assume that μ_g has a minimum value, say

$$\mu_{g,min} \equiv \frac{\mu_A + t_{min}}{2} \tag{40}$$

Then take

$$\sigma_g = \min\left(\frac{\mu_{g,min} - t_{min}}{4}, \sigma_A\right) \tag{41}$$

But to guarantee $\mu_g \geq \mu_{g,min}$ imposes yet another constraint on the maximum value of c. From (26):

$$\begin{aligned} \mu_g &= \mu_A - \sqrt{\frac{c}{a(a-c)}}\sigma_{Bg} \geq \mu_{g,min} \\ \Rightarrow \frac{c}{a(a-c)}\sigma_{Bg}^2 &\leq \left(\mu_A - \mu_{g,min}\right)^2 \end{aligned} \tag{42}$$

Comparing (42) with (30) reveals that it is the same equation, with the same solution, provided that you change t_{min} to $\mu_{g,min}$ in the coefficient equations (33) and (34). Thus, a second upper bound on c is found; call it $c_{max\,2}$. Then

$$c_{max} = min(c_{max\,1}, c_{max\,2}) \tag{43}$$

Another interesting quantity to solve for is the variance due to the outliers. This comes from (23) as

$$\sigma^2_{outliers} = c\widehat{U}^2 \tag{44}$$

Invoking (27) yields

$$\sigma^2_{outliers} = \frac{a-c}{a}\sigma^2_{Bg} = \frac{a-c}{a}\left(\sigma^2_B - (a-c)\sigma^2_g\right) \tag{45}$$

If want to find the minimum outlier variance, note that (45) is a quadratic function of c where the c^2 term has a negative coefficient. But c is an integer in the range $[1, c_{max}]$, so this means that the minimum of (45) occurs at one of those endpoints (i.e. either at 1 or c_{max}).

Computer algorithm

`Benchmark` implements the model above. It's overall goal is to determine the minimum fraction of the observed variance that must be attributable to outliers. If that fraction exceeds a threshold, then a warning is issued.

`Benchmark` starts off with knowing a, μ_B, and σ_B. It skips considering the outlier model if $a < 16$ or $\sigma_B = 0$.

If it passes that test, then it calculates μ_A and σ_A using (7) and (8). Next it takes $t_{min} = 0$ (which is the most conservative choice possible—it maximizes the ability of the Gaussian part to explain the measured variance without having to appeal to outliers). Then it calculates σ_g using (41)(39). Then it solves for c_{max} using (43). It skips considering the outlier model if $c_{max} < 1$.

Next it solves for the minimum variance caused by the outliers using (45) with c equal to 1 and c_{max}, and taking the minimum of those results; call it $\sigma^2_{outliers\,,min}$. If the ratio $\sigma^2_{outliers\,,min}/\sigma^2_B$ exceeds 1%, then a warning of some type is issued.

So, what does this outlier model produce for the "nanobenchmark" code in Listing 3? The standard deviation warnings look like

```
--block sd values MAY NOT REFLECT TASK'S INTRINSIC VARIATION
--guesstimate: environmental noise explains at least 100.0% of the measured sd
----------
--action sd values ALMOST CERTAINLY GROSSLY INFLATED by outliers
--they cause at least 99.60022873987793% of the measured VARIANCE according to a equi-valued outlier model
--model quantities: a = 67108864, muB = 1.395522860870968, sigmaB = 0.0013859776344426547, muA =
2.079491109953773E-8, sigmaA = 1.6918672295442562E-7, tMin = 0.0, muGMin = 1.0397455549768865E-8, sigmaG =
2.5993638874422163E-9, cMax1 = 998962, cMax2 = 252560, cMax = 252560, cOutMin = 252560, varOutMin =
1.9132546611046498E-6, muG(cOutMin) = 1.0397473789305775E-8, U(cOutMin) = 2.773147736700622E-6
```

The first part of the warning about the block sd values is produced by the noise floor measurement described above. In the run shown above, the noise floor explains 100% of task's sd. But in two earlier runs, not shown here, it only explained around 15% and 30% of task's sd, which illustrates the unrepeatability of this approach.

The second part of the warning about the action sd values is produced by the outlier model described in this section. Note that $c_{max} = 252{,}560$ is vastly smaller than $a = 67{,}108{,}864$, so the outliers are truly rare events. Furthermore, note when evaluated at $c = c_{max}$, then $\mu_g \approx \mu_{g,min} \approx 1.04 \times 10^{-8} \approx \mu_A/2$ and $U \approx 2.77 \times 10^{-6} \approx 266\mu_g$. In other words, the outliers truly are much larger than the typical Gaussian times. If these outliers are actually caused by context switches, and if the context switch rate reported above is accurate, then a better value to use would be $c = 2{,}500 \approx c_{max}/100$. Then from the $c \ll a$ result of (27),

U should increase by about 10 times to about 28 microseconds, which is about 56,000 clock cycles. It would be interesting to see if that value of U jives with other measurements of the impact of context switches.

Confidence intervals

Interpretation

The frequentist interpretation of confidence intervals is that if you repeat the confidence interval estimation procedure many times, p is the probability that any given confidence interval contains the true value. Other interpretations of probability (e.g. Bayesian) interpret confidence intervals differently. *However, it is always wrong to say that p is probability that the true value lies inside the interval.* This is subtly false: the true value has no uncertainty, even if it is unknown; it is either inside or outside a given interval. The only uncertainty is with the estimation procedure.

Relation with spread of measurements

Do not confuse the spread indicated by confidence intervals of the statistics with spread in the measurements themselves—they are independent quantities. For example, the confidence interval for both the mean and standard deviation could be very small (i.e. those values are precisely known), but the standard deviation could still be large relative to the mean (so the measurements themselves will be very scattered).

Alternatives

Confidence levels are not the only way to see if the means, say, of two populations are distinguishable. Statistical tests like Welch's t test could also be used. Unfortunately, many of these tests (e.g. Welch's) assume that populations have a Gaussian PDF, an assumption that can be suboptimal in benchmarking.

Furthermore, although simple statistics like the mean may have standard statistical tests, other statistics such as the standard deviation may not.

Gaussian PDF issues

The most common assumption when modeling execution times is that the PDF is a Gaussian. This is fundamentally appealing because of the Central Limit Theorem, but it is also practical since so many analytical results exist for Gaussian PDFs. Indeed, these reasons are why the outlier model above took the "non-outlier" execution times as being Gaussian distributed.

Nevertheless, the statistics section asserted that "it [bootstrapping] may yield more narrow and accurate confidence intervals than if you make wrong assumptions about the PDF (e.g. that it is Gaussian)". Furthermore, at several points above I cautioned against assuming a Gaussian PDF in the benchmarking context. This section will justify those remarks by showing how a Gaussian model can yield inferior results compared to nonparametric bootstrapping.

One problem with assuming a Gaussian PDF is that it implies negative execution times: even if the mean is positive, there is some part of the left tail that eventually extends below zero. But negative execution times are impossible, so this is clearly an error at some (possibly minuscule) level. In contrast, nonparametric bootstrapping never involves unphysical values (because it only uses resamples from the original measurements).

One scenario where a Gaussian PDF assumption produces less accurate results than bootstrapping is when discrete time errors dominate. Specifically, suppose that a computer's time measurement behaves as follows: if an event occurs inside the semi-open interval $[q\,dt, (q+1)\,dt)$, where q is an integer and dt is some fixed quantum of time, then it is measured as occurring at time $q\,dt$. Further, suppose that the task's real execution time has a PDF that is symmetric about $(q+1)\,dt$, and that its standard deviation is $\ll dt$. Then task's measured times will essentially follow a binary distribution: ~50% of the measurements will be $q\,dt$, the other ~50% will be $(q+1)\,dt$. The mean of this binary distribution is $(q+0.5)\,dt$, and its standard deviation is $0.5\,dt$. If you compare this binary distribution versus a Gaussian with $mean = \mu = (q+0.5)\,dt$ and $sd = \sigma = 0.5\,dt$, the Gaussian is clearly far more spread out in its range of likely values. This will be reflected in wider confidence intervals computed using this Gaussian than confidence intervals computed using bootstrapping. (Again, bootstrapping's resampling from the original measurements will tend to reproduce the binary distribution more closely.)

Task code: Callable versus Runnable

In the context of discussing task code for my benchmarking framework, I claimed "it is slightly easier to prevent DCE with a `Callable` than a `Runnable`".

That comment is perfectly true, however, there is one other consideration: the `Callable` DCE prevention rule might generate more garbage than if the task was written as a `Runnable`, and this could distort the benchmark. This happens if `Callable.call` returns a new object for each invocation, and if it is executed many times during benchmarking. In contrast, `Runnable.run` merely need store the effects of its computations in some internal state, which could be something very lightweight like a primitive field, so that no new objects are created during benchmarking. *This distinction is only important when writing the lightest weight microbenchmarks, and is probably irrelevant when benchmarking real applications.*

An example will make this obvious. Consider modifying Listing 1 in Part 2 so that it now benchmarks the computation of the 5th Fibonacci number:

Listing 4. Callable benchmark of the 5th Fibonacci number

```
public static void main(String[] args) throws Exception {
    Callable<Integer> task = new Callable<Integer>() { public Integer call() { return fibonacci(5); } };
    System.out.println("fibonacci(5): " + new Benchmark(task));
}
```

The 5th Fibonacci number takes vastly less time to compute than the 35th: my configuration found a mean execution time of 34.249 ns (confidence interval < 13 ps around this mean). In fact, each benchmark measurement executed `task` 33,554,432 times in order to achieve the default cumulative execution time goal of at least 1 second. That means that over 33 million `Integer` instances were created during each measurement. To see how much of an impact all this object creation and garbage collection has, lets rewrite the above using a Runnable:

Listing 5. Runnable benchmark of the 5th Fibonacci number

```
public static void main(String[] args) throws Exception {
    Runnable task = new Runnable() {
        private int sumOfResults = 0;
        public void run() { sumOfResults += fibonacci(5); }
        @Override public String toString() { return "sumOfResults = " + sumOfResults; }
    };
    System.out.println("fibonacci(5): " + new Benchmark(task).toStringFull());
}
```

Compared to Listing 4, the `Runnable` version is slightly longer (as it inevitably is, except that it has no generics—this is one slight advantage with writing your task as a `Runnable`). For the results, I find a mean execution time of 32.871 ns (confidence interval < 3 ps around this mean). So, the `Callable` version is slower than the `Runnable` version.

But it is only slower by ~4.2%. I am shocked at how small of an effect that is: I was expecting it to be much larger. These results suggest that the JVM is doing something really clever like `Integer` instance recycling instead of new instance creation. To test this, I modified the above code to use `Random.nextInt` instead of `fibonacci(5)`. The results are that the `Callable` version now has a mean of 49.685 ns, while the `Runnable` version has a mean of 34.972 ns, so the `Callable` version is ~42% slower than the `Runnable` version, which is a bit more like what I would expect. Thus, it looks like the JVM must be doing something clever with that Fibonnnaci code. But I am still puzzled: an ~15 ns overhead (49.685 – 34.972) for an `Integer` instance creation and garbage collection still seems low—are modern JVMs just that good, at least with simple objects and object graphs like in the code above?

Portfolio optimization

The section on portfolio optimization, naturally, focused mainly on the benchmarking aspects of the problem. However, I am worried that I will be besieged with emails from readers who want to know more about portfolio optimization. That, or else they are already knowledgeable, and find fault with the results! So, I decided to add some material here to hopefully stave off the deluge.

Dangers

With portfolio optimization, *a little knowledge is worse than none, because it might lead you to naively apply the theory, which is likely to lead to worse portfolios than if you did nothing at all*. For a layman-accessible account of portfolio optimization and some of its pitfalls, I suggest that you read *Chapter 5: Optimal Portfolio Allocations* in William J. Bernstein's book *The Intelligent Asset Allocator*. (In fact, that whole book is worth reading: it is the single best practical book on investing that I have come across.) For more advanced resources on portfolio optimization, the Web is your friend: seek and ye shall find.

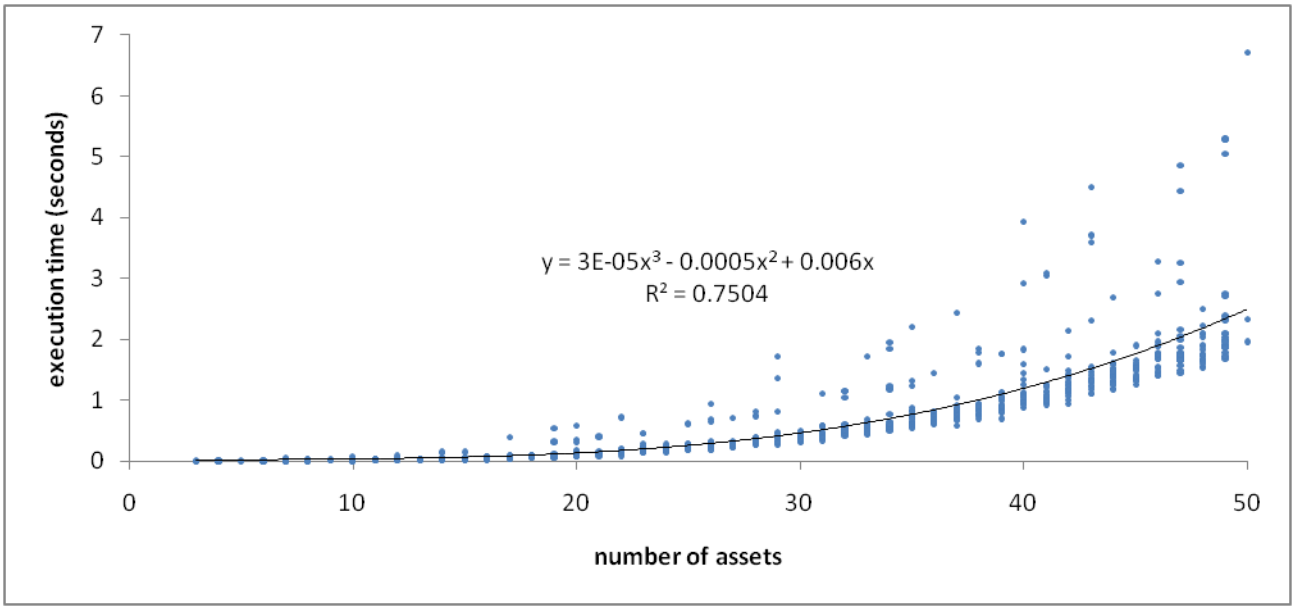
Data

One of the dangers that Bernstein mentions for portfolios constructed from indices of the major asset classes is that sufficient amounts of data must be used—over 10 years' worth—to account for a few cycles of mean reversion. (If you use too recent of data, your "optimized" portfolio will be dominated by yesterday's winners, but going forward, those are more likely to be tomorrow's losers.) There is a large body of literature on the long term mean reversion behavior of markets (as well as short term momentum effects); see, for example, the introduction of this paper for references or this talk. People routinely try to come up with strategies to exploit this. All I want to point out here is that *the data that used in the text was chosen solely because it is what I had readily available*. It is perfectly fine for the execution characteristics that I wanted to explore in the article. But it is almost certainly not the data that a skilled financial advisor would use to give you long term portfolio advice (e.g. he would use major asset class index funds instead). Corollary: any emails asking me what the "optimal" portfolios look like from my OEX data will be rudely ignored.

Constraints

Another danger that Bernstein mentions with naive application of portfolio optimization is that a few assets often end up dominating the portfolio, which is something that few sane investors would do. In order to prevent this, a common feature of most portfolio optimization software is that you can specify constraints on the asset weights. For example, if you are considering 10 assets, which if uniformly allocated would have 10% weight in each asset, you may wish to enforce the rule that no individual asset may be less than 5% or more than 20% of the portfolio. The `EfTask` class mentioned in the article can be run in either mode (with/without constraints). The results presented in the article have no constraints on the asset weights, simply because I did not want to bring up another non-benchmark related concept. But a real financial advisor would almost certainly impose constraints. Figure 3 shows the effect of using constraints on execution time and is analogous to Figure 6 in the article:

Figure 3. Portfolio optimization execution time



So, the execution time still grows approximately cubically as a function of the number of assets. But it grows faster, and has more scatter. This illustrates how certain sets of assets are very expensive to optimize in the presence of constraints.

Figure 4 shows the effect of using constraints on portfolio quality and is analogous to Figure 7 in the article:

Figure 4. Portfolio quality

The maximum Sharpe ratio with constraints imposed has decreasing extrema (on both the high and low side) as the number of assets increases. This is presumably due to the really good and really bad assets being increasingly diluted in weight as more assets get considered.

Length of time series

The length of the time series that is used has very little effect on the overall execution time unless truly massive lengths are present. Here, the time series is used to calculate the expected returns and covariance matrix inputs for the portfolio optimization. This calculation is vastly quicker than solving for the optimal

portfolio, at least for the lengths of time series that you commonly use. For example, my weekly OEX data has 238 elements in each time series. Doubling this length causes almost no change in execution time.

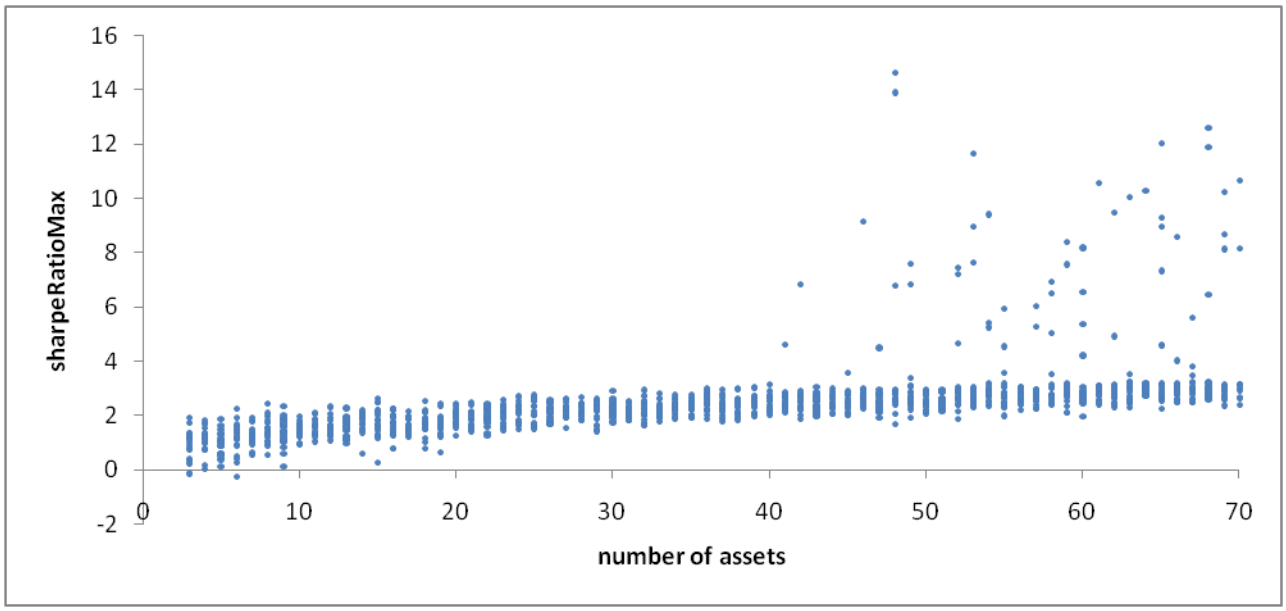
Sharpe ratio values

Figure 6 in Part 2 of the article has maximal Sharpe ratios of ~2 while Figure 4 above has maximal Sharpe ratios of ~1.6. Either value is much larger than the 3-year Sharpe ratio (as of 2007/12/31) for the S&P 100, which is 0.5475.

Why are my Sharpe ratio values so much larger than what is reported for the index? First, the article mentioned that dividends were not included in the return data, and if they were, my OEX Sharpe ratios would increase, making even larger of a gap. On the other hand, the Sharpe ratio for the S&P 100 weights the assets in the index by their market cap. In contrast, my OEX Sharpe ratios are calculated with the portfolio allowed to have either complete freedom (if no constraints used) or moderate freedom (if constraints imposed) in the asset weights, which clearly gives it a huge edge in increasing the *ex post* Sharpe ratio.

There is another interesting phenomenon relating to large Sharpe ratio values, namely, how they depend on the sampling period of the historical returns. Figure 5 again uses OEX data, but instead of weekly samples, it uses *monthly* samples:

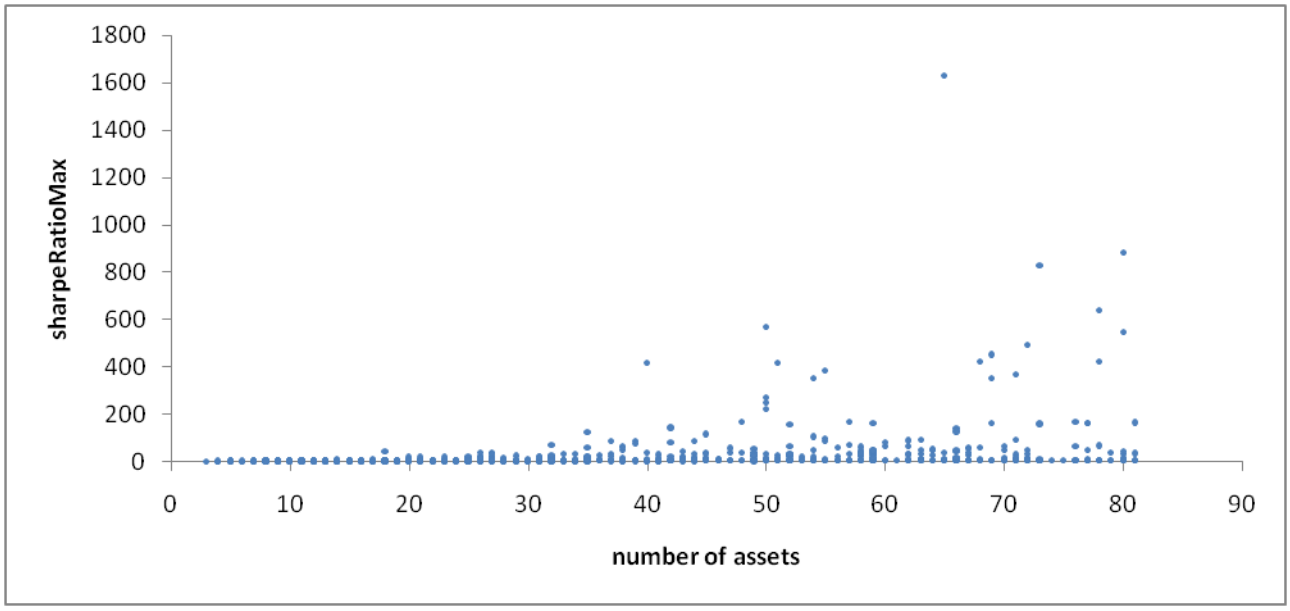
Figure 5. Portfolio quality (monthly samples)



No constraints on the asset weights were imposed, so this figure is comparable to Figure 6 in Part 2 of the article except that it uses monthly data. But the results have one difference: this graph has a fair number of huge outliers. There is not a fund manager in the world who wouldn't sell his first–born son for a Sharpe ratio of 5, let alone 14!

Think that those are spectacular results? Well consider Figure 6, which was obtained using *yearly* data for all non-tax-exempt Vanguard mutual funds that have existed over 1998-2007 (there are 81 such funds):

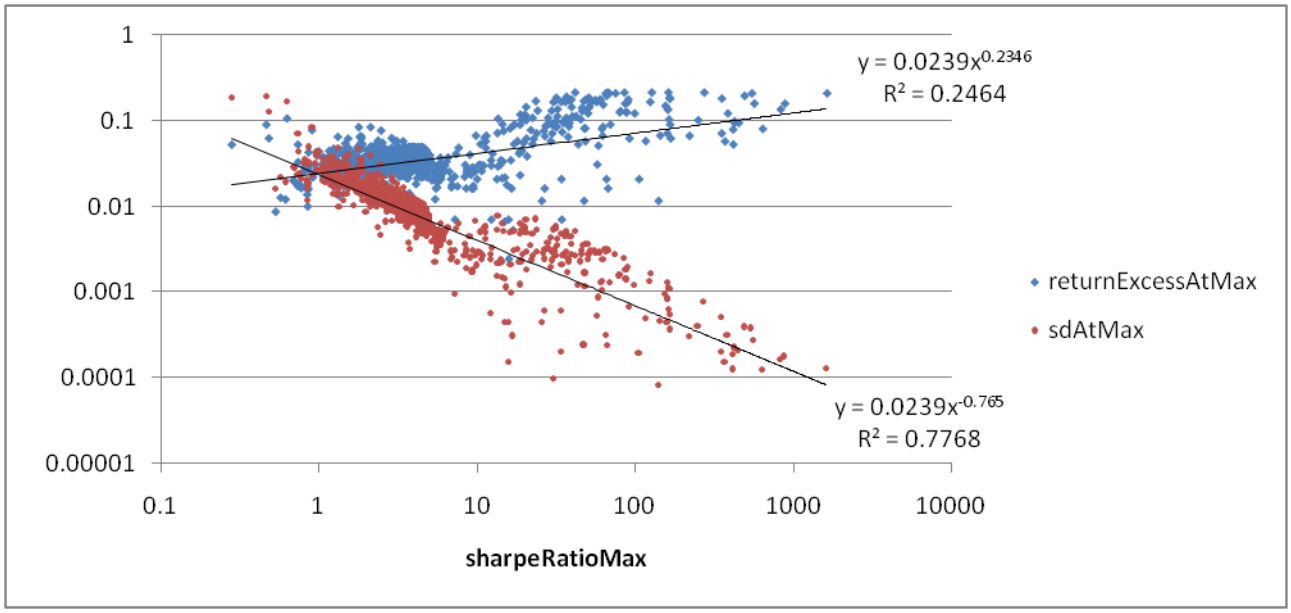
Figure 6. Portfolio quality (yearly samples, Vanguard funds)



Now we see outliers with astronomical values!

What causes this Sharpe ratio blowup as the sample frequency is decreased? It does not seem to be a defect in the WebCab portfolio optimization code. Instead, it appears to be caused by spurious covariances due to insufficient data. To see this, consider Figure 7, which uses that same Vanguard data as above, but this time takes the maximum Sharpe ratio for a set of assets as the independent variable, and plots both the excess return as well as the standard deviation that went into calculating that Sharpe ratio:

Figure 7. Excess Return and standard deviation as a function of Sharpe ratio



What this figure clearly shows is that *the really large Sharp ratios are primarily achieved by portfolios with extremely small standard deviation, as opposed to large excess return*. This is exactly what you would expect: the maximum excess return of the portfolio can be no better than that of the best asset's excess return, but the portfolio's standard deviation can go to zero by a fortuitous covariance matrix.

As the sampling frequency decreases, you get fewer data points in each historical time series. For example, for the 3 year OEX data with weekly sampling, each time series has $3 \times 52 = 156$ elements, but with

monthly samples only has $3 \times 12 = 36$ elements. And the Vanguard data only has $10 \times 1 = 10$ elements per time series.

What I think is happening is that as the number of elements per time series decreases, the chance greatly increases that you will find several time series that appear to be anticorrelated (or at least totally uncorrelated) with each other, and it is this low correlation that allows low standard deviation portfolios to be constructed. But these low correlations are bogus: they are caused by using insufficient numbers of data points, especially relative to the large number of assets being considered (100 for OEX, 81 for Vanguard).

Endnotes

[1] Get back to me if you know of a solution to this problem. I was unable to find anything in a web search; the closest result that might be relevant is the order statistic distribution function.

[2] Similarly, S_n^2 is a better estimator than S_{n-1}^2 for the variance because it has lower MSE even though S_n^2 is biased while S_{n-1}^2 is unbiased. In fact, S_{n+1}^2 has even lower MSE than S_n^2. I am not sure why S_{n+1}^2 is not commonly used; I could find no discussions of it on the web besides that wikipedia hyperlink...

[3] The details of how benchmarking is actually carried out is highly customizable; see `Benchmark`'s javadocs, as well as the `Benchmark.Params` inner class.

[4] Loop unrolling is probably the main optimization. The fundamental bitwise operations, however, cannot be eliminated.

[5] `perfmon.msc` with `system/context switches` or `thread/context switches`. Access this via `Control Panel → Administrative Tools → Performance`, then click on the "+" icon and select either `System` or `Thread` as the Performance object, then select `Context Switches/sec` from the list. See also this tutorial.

[6] See the bullet point on the resolution of `getCurrentThreadCpuTime` in the Execution time measurement section of Part 1.

[7] See the `Benchmark.Params.sdFractionThreshold` field.