Robust Performance Testing on the Cheap

Jeffreys Copeland and Haemer

January 2011

Abstract

We find ourselves impaled on a common software-performance-testing dilemma: how to measure performance meaningfully. To solve our problem, we invent a statistical test. Out of this, we get an automated performance-testing scheme that's cheap and easy to implement.

The Dilemma

Software changes constantly. It's easy to change performance by accident while trying to add new features or fix bugs. When that happens, we want to know. (By performance, we usually mean speed, though the same problem arises when measuring other performance limits, such as memory usage.)

A typical approach, borrowed from hardware testing, is to try to nail down the test environment, eliminating as much noise as possible, then run a small, defined set of performance tests, recording the numbers and tracking changes to those numbers over time.

This approach has a handful of problems:

1. **Run-to-run variation.** Multitasking computers run many processes simultaneously. These processes compete for resources with your tests; the resulting variation is hard to control for.

   The software you're testing may even have several components that run simultaneously, and their order and priorities may not be something you can (or even should) control during testing.

   Both of these can lead to a fair amount of run-to-run variation.

2. **Limited scope.** For complex software, the more kinds of test cases you can run, the more kinds of changes you can detect. A requirement to make the environment and test target as precise as possible

blocks this. What's more, dedicating hardware to specialized, tightly controlled, performance testing makes it unavailable for other testing.

3. **Multiple platforms.** Successful software often presses to be portable. As soon as your customers have a wide variety of machine types and configurations, testing performance on a single configuration is like looking under the lamp-post for your keys because the light's better there.

4. **Moore's law.** The computer you can buy today is faster, and has more memory and a bigger disk, than a computer you bought a year ago at the same price.

   If your test measurements require a configuration that you established two years ago, an upgrade will let you do more tests in less time, but you'd have to figure out how to compare today's apples to yesterday's oranges. This problem won't go away soon. [Moore 1965][1]

On one horn, you want to define the test environment precisely. On the other, this can create artificial numbers that are less-and-less relevant as time goes on.

This is particularly problematic for software performance testing because the ecosystem in which software lives metamorphoses so often and so dramatically.

We'd like a performance test methodology that's robust in the face of these sorts of changes.

Two Kinds of Performance Change

It helps to break performance changes into two broad categories.

The first is changes that change everything down uniformly.

Sometimes, someone really does make a change that speeds everything up by a factor of two. Porting compute-bound tests to hardware that's half as fast has the opposite effect.

Let's put changes that slow things down (or speed them up) monotonically into this group, too. An example might be something that cuts down the run time of every test to the logarithm of the original time.

[1] Gordon Moore, "Cramming more components onto integrated circuits," *Electronics* **38** (1965)

The second is everything else — changes that effect some tests, but leave others untouched.

For example, when printer-language developers tweaks PDF printing to make it more efficient, only the PDF-printing-performance tests speed up. If engineers do their work on machines with fast disks and a fast file system, but the target hardware has slow disks and an operating system with particularly inefficient disk handling, I/O-bound tests on the target may prove unexpectedly slow compared to compute-bound tests.

We will focus on detecting this second class of changes.

The Solution: Test Ranks

Consider, for a moment, the ranks of test times instead of raw times. If we rank time-to-completion of ten tests, a uniform slowdown or speedup will leave these ranks unchanged. All other things being equal, a collection of tests that rank *1 2 3 4 5 6 7 8 9* on an Intel Atom netbook should still rank *1 2 3 4 5 6 7 8 9* on a supercomputer: the first test's still faster than the last.

But if tests that have always ranked *1 2 3 4 5 6 7 8 9* abruptly show up ranked *9 8 7 6 5 4 3 2 1*, we have reason to believe something is afoot. This is an advantage of looking at ranks instead of raw numbers.

Ranks also eliminate some run-to-run noise effects. In unchanging environments, run times like these:

8.3, 5.24, 1.2, 18.5

are transformed into ranks like these:

3, 2, 1, 4

and are indistinguishable from these:

9.1, 5.0, 0.9, 18.0

In both, the third test is the fastest, the last one is the slowest.

Not all noise effects go away, though. Suppose, for example, we see this on one run

8.1, 7.9, 1.2, 18.5
[3, 2, 1, 4]

and this on a second

8.0, 8.2, 1.2, 18.5
[2, 3, 1, 4]

should we think they're different, or not? Maybe the first two tests are really the same speed and the difference is noise. Can we make a statistic that helps us measure that?

Why yes, we can.

The Null Hypothesis: All Ranks Are Equivalent

As a first step, let's look at what happens when every test is really the same speed — when every difference is due to system noise.

On one end of the spectrum, if tests ranked $1 \ldots N$ in the first run always have the same relative times, no matter how many times I run them, then all N ranks will be stable from run to run.

Suppose, however, the tests are just as likely to come up in any other position in a second run — the order is random and $N \ldots 1$ is just as likely as $1 \ldots N$. In this case, if we do two runs, how many tests are likely to have the same rank in both runs?

Answer: one.

Among all $N!$ equally probable arrangements for the second run, the test that's rank 1 in the first has rank 1 in $(N-1)!$ of them. The same holds for each of the N original ranks, so the total number of stable ranks over all $N!$ possible permutations is $N(N-1)! = N!$. If μ_S is the expected number of stable ranks,

$$\mu_S = \frac{N!}{N!} = 1 \tag{1}$$

(See the Appendix for an alternate derivation, using subfactorials.)

But how often should I expect to get 2 stable points instead? Or 3? What's the distribution?

The Distribution of Stable Ranks is Approximately Poisson

Imagine a lot of different test cases, all of which run in the same amount of time. All rank differences are caused by noise.

If I have 10,000 equivalent tests, whose run time varies at random, from run to run, then the chance that the any test in the first run will have the same rank time in the second is only 0.0001, but there are 10,000 different ranks.

This is a textbook setup for a Poisson distribution. The probability of seeing exactly k events, if there are, on average, λ, is given by

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{2}$$

For our case, the expected number of stable ranks is $\lambda = 1$. This means

$$P(\text{exactly S stable ranks}) = f(1;S) = \frac{e^{-1}}{S!} \tag{3}$$

The chance that a random rearrangement will leave only 1 test with the same rank it started with — exactly 1 stable rank — is $P(1) = \frac{1}{e} \approx 0.37$. Ditto for the chance of no stable ranks.[2]

The chance of finding exactly 2 stable ranks is half that.

A hallmark of the Poisson distribution is that the variance and the mean are equal. In our case, Poisson-distributed, stable-rank counts would show $\sigma_S^2 = \mu_S = \lambda = 1$.

This is also true if I have 100,000 equivalent tests; however, on the other end of the scale, we hit a special case.

If, however, instead of 10,000 tests, you have only one, the mean number of stable ranks is still 1. No matter how many times you run the test, it's still the fastest. Or the slowest, depending on whether your glass is half empty or half full. But the variance of this count is 0. $\mu_S = 1; \sigma_S^2 = 0$

Exactly one test? Exactly one stable rank, every single time.

The distribution of stable ranks for $N = 1$ isn't even close to Poisson.

So, if we have a lot of equivalent tests, the number of stable ranks is Poisson distributed, and if we only have one test, it's not. How about for numbers of tests in between 1 and 10,000? How fast does the distribution of the number of stable ranks approach a Poisson?

For starters, we might ask how large N must be for the variance to begin to approach the mean.

A little algebra invoking subfactorials yields a surprising result: $\sigma_S^2 = \mu_S = 1$ for every $N > 1$.

Okay, it surprised us. (For details, see the Appendix.)

[2]This provides another proof of the well-known limit $\lim_{n\to\infty}(\frac{n_!}{n!}) = \frac{1}{e}$ (for special values of "well-known").

The Poisson Approximation Works for Small N

Since the mean and variance are identical for all values of $N > 1$, we need to ask about the approach to "Poisson-ness" in a different way.

First, you can see that none of our distributions will really be a Poisson. The tail of a true Poisson stretches rightwards to infinity: in a Poisson, $P(S) > 0$ for all values of $S > 0$. Not so for stable ranks. For a collection of tests of size N, $P(S) = 0$ for all $S > N$. When you re-run 10 tests, you won't ever get 11 that have the same rank they did the first time.

We'll illustrate with a simulation. Here's example output for $N = 5$.

```
iterations = 1000, N = 5
mean(nr of stable points) = 0.968
var(nr of stable points) = 0.926976
observed counts: 366 396 154 78 6 0
expected counts (Poisson): 368 368 184 61 15 3 1
```

For each trial we rearranged 5 ranks at random and then measured the number of stable ranks. In 1000 trials, 366 of trials ended with no stable points, 396 trials had exactly one, and no trial had all five. In a true Poisson distribution, we'd have expected 368 with no stable points, 368 with exactly one, and 3 with all five.

And one with six. Whoops!

Still, the observed distribution looks, by eyeball, a lot like a Poisson, even at this, small value of N.

A χ^2 goodness-of-fit test detects a difference, but also suggests a way to ask the question we started with: how large does N need to be for the difference to be undetectable with a χ^2 test?

We don't have an analytic solution. In our simulations, however, when $N > 10$ even a χ^2 test is fooled: we cannot distinguish the distribution of stable ranks from one predicted by a true Poisson.

Even with quite small collections of tests of similar speed, a Poisson distribution with $\lambda = 1$ does an adequate job of predicting the number of stable ranks from run-to-run.

In Real-Life, Not All Tests Have The Same Ranks

A more typical situation might be this:

A set of fast tests form a cluster, c_1, which differ in their order only because of system noise. The next fastest group of tests, c_2, are all slower than the tests in c_1, but are, again, equivalent to one another.

Next, an individual test, t_1, runs slower than all the tests in c_2, but faster than the tests in a third cluster, c_3.

The remaining tests either arrange themselves into performance clusters or are singletons with unique orders; their run times arrange themselves on an axis like this:

$$c_1 < c_2 < t_1 < c_3 < t_2 < c_4 < t_3 < t_4 < \cdots \tag{4}$$

What is the expected number of stable ranks for the entire collection? If there are T tests with a unique order and C clusters, then

$$\mu_S = \sum_{i=1}^{C} \mu_{c_i} + \sum_{i=1}^{T} \mu_{t_i} = C + T \tag{5}$$

Since each the number of stable ranks for cluster c_n has a variance of 1 but each unique test has a variance of 0,

$$\sigma_S^2 = C \tag{6}$$

we have, as an upper limit,

$$\sigma_S^2 \leq \mu_S. \tag{7}$$

Even a pair of unrelated tests with equivalent run times, modulo noise, make a cluster, so if we're collecting data from enough different tests, with few singletons, we may have

$$\sigma_S^2 \simeq \mu_S \tag{8}$$

If the number of tests with unique run times is small, we can approximate the overall distribution of stable points as a sum of independent Poisson distributions with identical distributions. Because the sum of Poisson distributions is, counter-intuitively,[3] a Poisson distribution[4] the distribution should be approximately this:

[3] F.W. Stahl, personal communication.

[4] Fredrick Mosteller and R.E.K Rourke, *Sturdy Statistics: Nonparametric and Order Statistics*. Addison-Wesley, 1973. If you sum *enough* Poisson distributions, then λ becomes large, and the distribution becomes approximately Normal, as the Central Limit Theorem predicts.

$$P(\text{S stable ranks}) \simeq f(S; C+T) = \frac{(C+T)^S e^{-(C+T)}}{S!} \tag{9}$$

The average number of stable points will be $\mu_S = C+T$, and the standard deviation of that number, $\sigma_S \leq \sqrt{C+T} = \sqrt{\mu_S}$.

A Recipe for Cheap, Automated, Performance-Regression Testing

How can you put this into practice? Here's a recipe.

1. **Establish a baseline.**

   Run a large number of different test cases through the system.

   Unit tests, functional tests, conformance tests, ... everything's grist for the mill. Indeed, the more diverse your tests, the better, because they will be affected differently by performance changes in different parts of the system under test.

   Rank all tests by time-to-completion.

   Don't create separate performance testing suite, or even do a separate run, just time the ones you're already doing.

2. **Find a mean and variance for the statistic.**

   Run them again a few times, and look to see how many ranks stay the same from run-to-run. Noise should produce different stable ranks in different runs, but you can ignore that detail. You're only looking for a total.

   If, however, there are, say, 100 stable ranks, then that's your expected value and an upper bound to the approximate variance. The standard deviation is the square root of that: 10.

3. **Use the statistic as a sentinel**

   Have your test software report this number every day. You should expect something between 80 and 120 stable ranks ($\mu \pm 2\sigma$).

   Or pour the code being tested onto a different platform: a shiny, new Intel Mac Air instead of an archaic G3 iBook or, perhaps a quad-core Athlon box running Linux instead of an five-year-old IBM

Thinkpad, running Windows XP. Then run and rank the same tests. The run times will change dramatically, and even non-linearly, but they shouldn't change wildly: you still expect between 80 and 120 stable ranks.

However, when the number of stable ranks drops to 50 or rises to 150, some aspect of performance has changed significantly. Time to raise a red flag and investigate further.

We've presumed, above, that the process is in place, but that need not even be true.

A customer complains of a dramatic, overall performance drop from an earlier release of two years ago? Run a wide-spectrum, performance-stability test on daily checkpoints from your source-code database to see which revisions produced big changes.

After you've found the neighborhood to look in, turn to your special, locked-down, carefully defined performance-testing platform for careful analysis, to identify the specific culprit.

This recipe automatically turns leftovers into food.

Summary

Automated software-performance testing may seem like it's thwarted by a Heisenberg-like Uncertainty Principle: the closer you look, the less your numbers mean.

One way around this is to fall back on a non-parametric approach that looks at ranks instead of precise times. If you use a large number of randomly chosen test cases, the number of stable ranks from run to run becomes Poisson distributed.

This offers a simple way to turn existing, large libraries of automated functional, integration, and system tests into platform-independent performance tests at little or no cost.

The test cases need not be related or carefully constructed. You don't need to know the internals of the tests themselves. The clusters of run times are formed by chance, not for any functional reason. The more clusters – the more tests that accidentally have the same run-times – the better. And it's as automated as all the rest of your tests since, ideally, it's built from all the rest of your automated tests.

Doing this doesn't tell you what's changed when you see a change — that requires further investigation — but it gives you an almost free sentinel for when you need to investigate.

You only need to start timing (and ranking) all the tests you're already running.

Appendix: Stable Ranks in Random Permutations

Definitions and Notation

A derangement is a permutation in which no element winds up in its original position. For example, with the numbers 12 69 7, the permutation 69 7 12 is a derangement, but 7 69 12 is not because 69 remains in the second position.

The subfactorial of N is the number of derangements of a collection of N, distinct elements.

There are several different notations for this function; we'll use Knuth's $N¡$ because it is less ambiguous than the alternative $!N$.

Conventions and a Useful Theorem

By convention, $0¡ = 1$, just as $0! = 1$. This makes a lot of subfactorial formulas consistent. However, note that $1! = 1$, but $1¡ = 0$. There's no way to derange a single element.

To reduce visual clutter, if we write an index-less sum, we mean the index ranges from 0 to n, in whichever direction is convenient. That is,

$$\sum k = \sum_{k=0}^{k=n} k = \sum_{k=n}^{k=0} k \tag{10}$$

We will use this theorem, presented without proof,

$$n¡ = (n-1)¡ + n \cdot (-1)^n \tag{11}$$

or its equivalent

$$n¡ - (-1)^n = n \cdot (n-1)¡ \tag{12}$$

which we lift from Graham, Knuth, and Patashnik [1994][5]

[5] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 1989; second edition, 1994.

Identities We'll Use

We shall use three identities about binomial coefficients.

$$\binom{n}{r} = \binom{n}{n-r} \tag{13}$$

$$r\binom{n}{r} = \binom{n}{n-r+1}(n-r+1) \tag{14}$$

$$(r+1)\binom{n}{r+1} = \binom{n}{r}(n-r) \tag{15}$$

Finally, a special case of the binomial theorem

$$f(x) = \sum_0^n \binom{n}{k} x^k = (1+x)^n \tag{16}$$

yields both this

$$\sum \binom{n}{k}(-1)^k = 0 \tag{17}$$

and this

$$f'(-1) = \sum k\binom{n}{k}(-1)^{(k-1)} = 0 \tag{18}$$

Combining (12), (16), and (17) also gives us this

$$\begin{aligned} \sum k\binom{n}{k}(-1)^{(n-k)} &= \sum (n-r)\binom{n}{n-r}(-1)^r \\ &= \sum n\binom{n}{n-r}(-1)^r - \sum r\binom{n}{n-r}(-1)^r \\ &= n\sum \binom{n}{r}(-1)^r + \sum r\binom{n}{r}(-1)^{(r-1)} \\ &= 0+0=0 \end{aligned} \tag{19}$$

Stable Ranks over All Permutations

Theorem: The total number of stable ranks over all permutations on n elements is

$$S_n = \sum_{k=0}^{n} k \binom{n}{k} (n-k)¡ = n! \tag{20}$$

Proof: Consider all permutations on n elements. Among these, let the number of permutations with exactly k stable ranks be $S_n(k)$.

This number is the product of the number of ways we can choose exactly k ranks to hold in fixed positions, times the number of ways we can rearrange the remaining elements so that none is in its original place. That is,

$$S_n(k) = \binom{n}{k} (n-k)¡$$

and, adding up all cases,

$$n! = \sum S_n(k) = \sum \binom{n}{k} (n-k)¡ \tag{21}$$

To get the total number of stable ranks in all permutations, we multiply each term of this sum by k, the number of stable ranks it represents.

$$\begin{aligned} S_n &= \sum_{k=0}^{n} k S_n(k) = \sum_{k=0}^{n} k \binom{n}{k} (n-k)¡ \\ &= \sum_{k=1}^{n} k \binom{n}{k} (n-k)¡ = \sum_{r=0}^{n-1} (r+1) \binom{n}{r+1} [n-(r+1)]¡ \end{aligned} \tag{22}$$

Transforming with (14) gives us

$$= \sum_{r=0}^{n-1} \binom{n}{r} (n-r)[n-(r+1)]¡ \tag{23}$$

From (11), we have

$$(n-r)¡ = (n-r) \cdot [(n-r)-1]¡ + (-1)^{(n-r)} \tag{24}$$

so

$$
\begin{aligned}
S_n &= \sum_{r=0}^{n-1} \binom{n}{r} [(n-r)¡ - (-1)^{(n-r)}] \qquad (25)\\
&= \sum_{r=0}^{n-1} \binom{n}{r} (n-r)¡ - \sum_{r=0}^{n-1} \binom{n}{r} (-1)^{(n-r)} \\
&= \left[\sum_{r=0}^{n} \binom{n}{r} (n-r)¡ - \binom{n}{n} (n-n)¡ \right] \\
&\qquad - \left[\sum_{r=0}^{n} \binom{n}{r} (-1)^{(n-r)} - \binom{n}{n} (-1)^0 \right]
\end{aligned}
$$

Invoking (20), (12), and (16), we get

$$= n! - 1 - \sum_{r=0}^{n} \binom{n}{n-r} (-1)^{(n-r)} + 1 = n! \qquad (26)$$

This also gives us the result we found in equation (1), but counting in a different way.

$$\mu_S = \frac{S_n}{n!} = 1 \qquad (27)$$

Variance of Number of Stable Ranks

Theorem: The variance of the number of stable ranks in permutations on N objects ($N > 1$) is

$$\sigma_S^2 = 1 \qquad (28)$$

Proof: Begin with the standard familiar formula

$$\sigma_S^2 = E(S^2) - E^2(S) \qquad (29)$$

To get the first term, we consider the equation

$$W_S = n! E(S^2) = \sum_{i=0}^{n} k^2 \binom{n}{k} (n-k)¡ \qquad (30)$$

As before, we transform indices

$$= \sum_{k=1}^{n} k^2 \binom{n}{k} (n-k)_¡ = \sum_{r=0}^{n-1} (r+1)^2 \binom{n}{r+1} [n-(r+1)]_¡ \tag{31}$$

from (14)

$$\begin{aligned} &= \sum_{r=0}^{n-1} (r+1) \binom{n}{r} (n-r)[n-(r+1)]_¡ \qquad (32)\\ &= \sum_{r=0}^{n-1} r \binom{n}{r} (n-r)[n-(r+1)]_¡ \\ &\qquad + \sum_{r=0}^{n-1} \binom{n}{r} (n-r)[n-(r+1)]_¡ \\ &= T_1 + T_2 \end{aligned}$$

T_2 is easy: we just walk backwards, repeating step (14) and retransforming indices in reverse.

$$\begin{aligned} T_2 &= \sum_{r=0}^{n-1} \binom{n}{r} (n-r)[n-(r+1)]_¡ \qquad (33)\\ &= \sum_{r=0}^{n-1} (r+1) \binom{n}{r+1} [n-(r+1)]_¡ \\ &= \sum_{k=1}^{n} k \binom{n}{k} (n-k)_¡ \\ &= \sum_{k=0}^{n} k \binom{n}{k} (n-k)_¡ \\ &= n! \end{aligned}$$

(The last step from equation 20).

T_1 isn't hard either. We invoke (11), (19), and (18).

$$\begin{aligned} T_1 &= \sum_{r=1}^{n-1} r \binom{n}{r} (n-r)[n-(r+1)]_¡ \qquad (34)\\ &= \sum_{r=0}^{n-1} r \binom{n}{r} [(n-r)_¡ - (-1)^{(n-r)}] \end{aligned}$$

$$
\begin{aligned}
&= \sum_{r=0}^{n-1} r\binom{n}{r}(n-r)¡ - \sum_{r=0}^{n-1} r\binom{n}{r}(-1)^{(n-r)} \\
&= \left[\sum_{r=1}^{n} r\binom{n}{r} - n\binom{n}{n}0¡\right] - \left[\sum_{r=1}^{n} r\binom{n}{r}(-1)^{(n-r)} - n\binom{n}{n}(-1)^0\right] \\
&= (n! - n) - (0 - n) \\
&= n!
\end{aligned}
$$

so

$$
\begin{aligned}
W_n = n!E(S^2) &= T_1 + T_2 = n! + n! = 2n! \qquad (35)\\
E(S^2) &= 2.
\end{aligned}
$$

And finally

$$\sigma_S^2 = E(S^2) - E^2(S) = 2 - 1 = 1. \qquad (36)$$