Benchmark It!

Ryan James Spencer

Performance is something users *feel*. Maybe you're in an organization that doesn't care about performance, and you think they should. Perhaps there are lofty claims that performance *does* matter, but loose arguments for one approach over another filled with baseless claims like "X is *fast*!". The only way to get past this conjecture is with numbers, and we collect numbers with benchmarks.

Benchmarking is the act of measuring latency or throughput of some component regardless of size. Profiling is a means to explore the constituent parts of where code spends its time. There is a dizzying array of variables we can tweak to impact the performance of user experience on modern platforms, but benchmarks should come first before we begin profiling. With benchmarks, we get reproducibility, enabling others to test our claims and verify our results for their use cases. As we generate measurements for targets, we can store the results away so we can visualize the data however we please or compare particular runs tied to specific code changes. In this sense, benchmarks provie us a historical context of performance.

Latency is the duration of time the component in question takes to complete. Throughput, on the other hand, is the amount of work completed in a window of time. You cannot trust the first number you get out of a single trial. Hence, we focus on arithmetic mean averages or medians from multiple measurements. A server might have an average latency of 123 milliseconds, or a JSON parser might boast an average of 2.6GiB processed per second.

Cargo does have built-in support for benchmarking, and it works in a pinch but hides the influence of outliers on your results. This is handy if you want something you can quickly run and glance at to make improvements, but tests are reliable if they are low on noise.

To gain this insight, I strongly recommend criterion.rs. It does all the things `cargo-bench` does, along with providing additional statistical metrics to help us determine if a benchmark is worth trusting.

By default, criterion takes one-hundred measurements. It then tries to find a line that fits these measurements using linear regression. How well this line fits the data is designated by the metric R^2. The slope of this line, along with the mean and median, offer ways of viewing "center" for the data. If there are no outliers, then slope, mean, and median should be similar. Standard deviation is the dispersal of values around the mean, and MAD or Median Absolute Deviation is the same for the median. A high standard deviation or MAD might indicate a higher than expected level of noise. Similarly, if the R^2 is low, then the difference between timings is high. To get reliable tests, we want each iteration to be the same. There is bound to be *some* noise and differences between runs, but we are trying to find the values we are confident aren't merely aberrations.

You can add it to a project by adding the dependency to your Cargo.toml:

```
[dev-dependencies]
criterion = "0.3"
```

Then, in the same file we can add the benchmark:

```
[[bench]]
name = "benchmark_it"
harness = false
```

benchmarks live in distinct files under the `benches` directory in the root of your project, named the same as the name we gave in the manifest:

```
$ fd benchmark_it
benches/benchmark_it.rs
```

Imagine our crate is called `crate` and it exposes a public function named `function` that we want to measure, the most basic benchmark looks something like the following:

```
use criterion::{black_box, criterion_group, criterion_main, Criterion};
use crate::function;

pub fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("benchmark_function", |b| b.iter(|| function(black_box(20))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

and you can run this benchmark with `cargo  bench`. After the tests run you get output on the command line. If you have gnuplot installed you can also view an HTML report generated at `target/criterion/report/index.html` you can open in a browser. If you want access to the raw data, you can find that dumped as CSVs under `target/criterion/benchmark_function/{base,change,new}/raw.csv`. As you run the benchmarks, the base gets replaced with the last latest run, and the latest run is compared to visualize improvements or regressions.

Criterion's reports include explanations in reports, along with fantastic documentation outlining how the test harness works and how the statistics are calculated, including diagnostics in the output about when different outliers are detected.

To further reduce noise, you want to run on a machine that matches your target environment as much as possible, which depends on your domain in question. If you are running a server on a specific configuration, test the code on that hardware with that configuration. If you are building a CLI tool used by developers, it might make sense to have many benchmarks from commodity hardware that developers are using, such as laptops with little to nothing else running on the system. We might also want to benchmark on the fastest, quality hardware we can find to determine limits of what you could hope to attain given empirical results.

Benchmarks can explain performance under various loads. Input now comes into the picture. If you are a data-regulation compliant company, and I hope you are self-compliant if not, then generating data that has the same characteristics and cardinality of what you tend to expect is vital to feed into your benchmarks as expected "normal load" under the system. You also want to try to record pathological cases where the system is exceptionally slow under particular input. This isn't to say these pathological cases need to be your primary target for the profiling and optimization that comes later. You might want to improve the life of 99% of your users rather than worrying about an edge case that happens 0.001% of the time. However, pathological cases still give us insight into the limits of the component under measurement.

Independent of where you store your benchmarks, having them recorded for every commit, or possibly every master commit, can let you easily compare two changes using something like Andrew Gallant's tool cargo-critcmp. If you have different hardware to test, you can script checking out changes, running the benchmarks, and comparing the results. When making comparisons, make sure to minimize variables of change across the various measurements! You don't want one to have programs running in the background while the other was on a totally silent system, for example.

Andy Gavin talks about how they benchmarked the various display modes for the Sony Playstation during the making of Crash Bandicoot. He remarks how if he had not done this and taken the recommended mode at face value, it would have been subpar for their situation! This is precisely the kind of speculation that measurements help dispel. Performance matters and numbers help make performance tangible. Write benchmarks that work for *your* data and *your* setup! Arm yourself to the teeth with numbers and ensure they are *valid* numbers to be confident in your fight against lofty claims. *Valid* measurements are useful to the community, whether it's your local team or the open-source community as a whole. Write more benchmarks!

Subscribe