ECE459: Programming for Performance Winter 2021

Lecture 25 — Profiling

Jeff Zarnett & Patrick Lam *2020-11-24*

Profiling

Think back to the beginning of the course when we did a quiz on what operations are fast and what operations are not. The important takeaway was not that we needed to focus on how to micro-optimize this abstraction or that hash function, but that our intuition about what is fast and what is slow is often wrong. Not just at a macro level, but at a micro level. You may be able to narrow down that this computation of x is slow, but if you examine it carefully... what parts of it are slow?

If you don't use tools, then you end up guessing. You just make some assumptions about what you think is likely to be slow and try to change it. You've probably heard the famous quotation before, but here it is in its full form:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> – Donald Knuth

So going about this blindly is probably a waste of time. You might be fortunate and optimize a slow part[1] but we should really follow one of my favourite rules: "don't guess, measure!"[2] So, to make your programs or systems fast, you need to find out what is currently slow and improve it (duh!). Up until now in the course it's mostly been about "let's speed this up", but we did not take much time to decide what we should speed up (though you maybe did this on an assignment...?).

The general idea is, collect some data on what parts of the code are taking up the majority of the time. This can be broken down into looking at what functions get called, or how long functions take, or what's using memory...

There is always the "informal" or "ad-hoc" way of doing this; it works but it's often not the best plan. You probably know that when developing a program you can "debug" it without using any tools (e.g., `gdb`) by inserting print statements to the console or the log file. So when you enter function `foo` you print a nice little line on the console that say something like "entering function foo", associated with a timestamp and then when you're ready to return, a corresponding print function that says "exiting" appears, also with a timestamp.

I've used this approach myself to figure out what blocks of a single large function are taking a long time (such as a time when I found out that updating exchange rates was slow.). But this approach is not necessarily a good one. It's an example of "invasive" profiling—we are going in and changing the source code of the program in question—to add (slow!) instrumentation (log/debug statements). Plus we have to do a lot of manual accounting. Assuming your program is fast and goes through functions quickly and often, trying to put the pieces together manually is hopeless. It worked in that one example because the single function itself was running in the half-hour range and I could see that the save operation was taking twelve minutes. Not kidding.

Nicholas Nethercote wrote the "counts" tool[3] to process ad-hoc debug output and shows an example of using it to profile the size of heap allocations in Firefox. In the example in the README, he reports that while most allocations

[1] There is a saying that even a blind squirrel sometimes finds a nut.
[2] Now I am certain you are sick of hearing that.
[3] `https://github.com/nnethercote/counts`

are small, most memory is allocated as part of a large allocation. He provides a number of other examples where ad-hoc profiling is useful: how often are certain paths executed? how many times does a loop iterate? how many elements are in a hash table at a given location? And more! It's hard to write a general-purpose tool for these ad-hoc queries.

(Also like debugging, if you get to be a wizard you can maybe do it by code inspection, but that technique of speculative execution inside your head is a lot harder to apply to performance problems than it is to debugging. Trained professionals like Nicholas Nethercote use profilers, so you can too.)

So we should all agree, we want to use tools and do this in a methodical way.

Now that we agree on that, let's think about how profiling tools work

- sampling-based (traditional): every so often (e.g. 100ms for gprof), stop the system and ask it what it's doing (query the system state); or,
- instrumentation-based, or probe-based/predicate-based (sometimes, too expensive): query system state under certain conditions; like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling. You can read more about profiling in Chapter 5 (Profiling) of The Rust Performance Book [N+20]; it discusses heap profiling in much more detail than we do (or don't, more accurately).

If you need your system to run fast, you need to start profiling and benchmarking as soon as you can run the system. Benefits:

- establishes a baseline performance for the system;
- allows you to measure impacts of changes and further system development;
- allows you to re-design the system before it's too late;
- avoids the need for "perf spray" to make the system faster, since that spray is often made of "unobtainium"[4].

Tips for Leveraging Profiling. When writing large software projects:

- First, write clear and concise code. Don't do any premature optimizations—focus on correctness. Still, there are choices you can make that support performance at this stage, like using an efficient search or sort algorithm, if you know it's better and won't take additional effort.
- Once it's working, profile to get a baseline of your performance: it allows you to easily track any performance changes and to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Look for abnormalities; in particular, you're looking for deviations from the following rules:

- time is spent in the right part of the system/program;
- time is not spent in error-handling, noncritical code, or exceptional cases; and
- time is not unnecessarily spent in the operating system.

For instance, "why is `ps` taking up all my cycles?"; see page 34 of [Can06].

[4] `http://en.wikipedia.org/wiki/Unobtainium`

Development vs. production. You can always profile your systems in development, but that might not help with complexities in production. (You want separate dev and production systems, of course!) We'll talk a bit about DTrace, which is one way of profiling a production system. The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Userspace per-process profiling

Sometimes—or, in this course, often—you can get away with investigating just one process and get useful results about that process's behaviour. We'll first talk about `perf`, the profiler recommended for use with Rust. This is Linux-specific, though.

The `perf` tool is an interface to the Linux kernel's built-in sample-based profiling using CPU counters. It works per-process, per-CPU, or system-wide. It can report the cost of each line of code.

Webpage: `https://perf.wiki.kernel.org/index.php/Tutorial`

Here's a usage example on some old assignment code from a previous offering of the course:

```
[plam@lynch nm-morph]$ perf stat ./test_harness

 Performance counter stats for './test_harness':

       6562.501429 task-clock                #    0.997 CPUs utilized
               666 context-switches          #    0.101 K/sec
                 0 cpu-migrations            #    0.000 K/sec
             3,791 page-faults               #    0.578 K/sec
    24,874,267,078 cycles                    #    3.790 GHz                      [83.32%]
    12,565,457,337 stalled-cycles-frontend   #   50.52% frontend cycles idle     [83.31%]
     5,874,853,028 stalled-cycles-backend    #   23.62% backend  cycles idle     [66.63%]
    33,787,408,650 instructions              #    1.36  insns per cycle
                                             #    0.37  stalled cycles per insn [83.32%]
     5,271,501,213 branches                  #  803.276 M/sec                    [83.38%]
       155,568,356 branch-misses             #    2.95% of all branches          [83.36%]

       6.580225847 seconds time elapsed
```

Right, let's get started. We're going to use the blog post [Per16] as a guide; that source tells a more complete story of an example of using the profiler to optimize, but for now we are just interested in the steps.

The first thing to do is to compile with debugging info, go to your `Cargo.toml` file and add:

```
[profile.release]
debug = true
```

This means that `cargo build -release` will now compile the version with debug info (you can tell because it will say `Finished release [optimized + debuginfo] target(s) in 0.55s`; without this, we wouldn't get the part that says debug info so we can tell if it's correct. And we want it to be the release version that we're instrumenting, because we want the compiler optimizations to be applied. Without those, we might be trying to optimize things where the compiler would do it for us anyway.

The basic plan is to run the program using `perf record`, which will sample the execution of the program to produce a data set. Then there are three ways we can look at the code: `perf report`, `perf annotate`, and a flamegraph. We'll look at all of those, but in a live demo.

CLion. While we've seen how to use `perf`, it's not the only way. During development of some of the code exercises, I used the CLion built-in profiler for this purpose. It generates a flamegraph for you too, and I'll show that for how to create the flamegraph as well.

References

[Can06] Bryan Cantrill. Hidden in Plain Sight, 2006. Online; accessed 20-Janaury-2016. URL: http://queue.acm.org/detail.cfm?id=1117401.

[N+20] Nicholas Nethercote et al. *The Rust Performance Book*. Self-published, 2020. Online; accessed 2020-11-24. URL: https://nnethercote.github.io/perf-book/.

[Per16] Adam Perry. Rust performance: A story featuring perf and flamegraph on linux, 2016. Online; accessed 2020-11-01. URL: https://blog.anp.lol/rust/2016/07/24/profiling-rust-perf-flamegraph/.