Lecture 25 — Profiling

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 5, 2022

Remember the Initial Quiz

Think back: what operations are fast and what operations are not?

Takeaway: our intuition 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?

Premature Optimization

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

That Saying You Were Expecting

Feeling lucky?
Maybe you optimized a slow part.

To make your programs or systems fast,
you need to find out what is currently slow and improve it (duh!).

Up until now, it's mostly been about
"let's speed this up".
We haven't taken much time to decide what we should speed up.

Basic Overview

General idea:

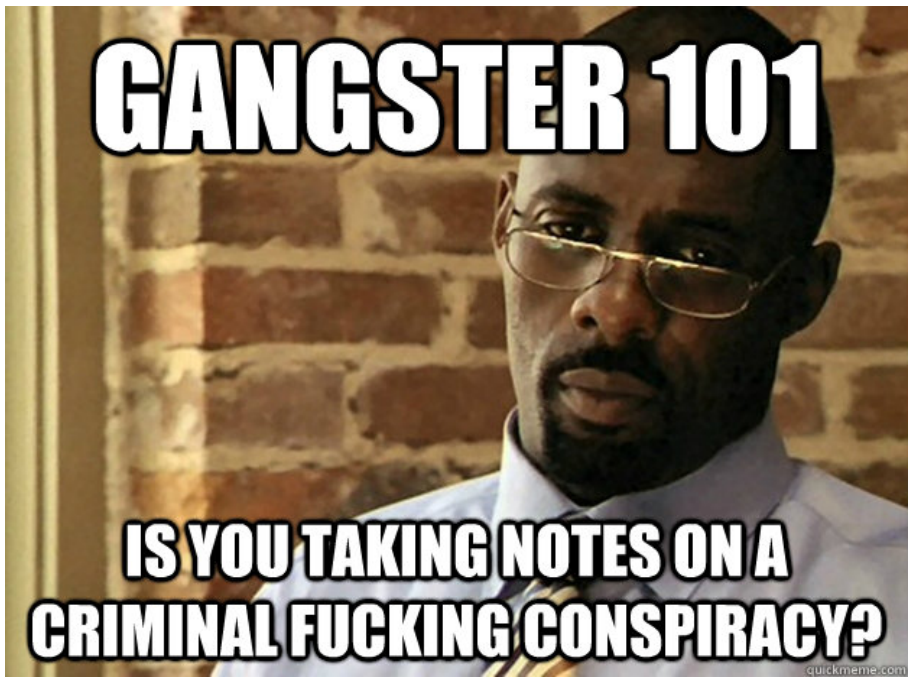
collect data on what parts of the code are taking up most of the time.

- What functions get called?
- How long do functions take?
- What's using memory?

There is always the “informal” way: “debug” by print statements.

On entry to `foo`, log “entering function foo”, plus timestamp.

On exit, log “exiting foo”, also with timestamp.



Console Profiling

Kind of works, and [JZ] has used it... But this approach is not necessarily good.

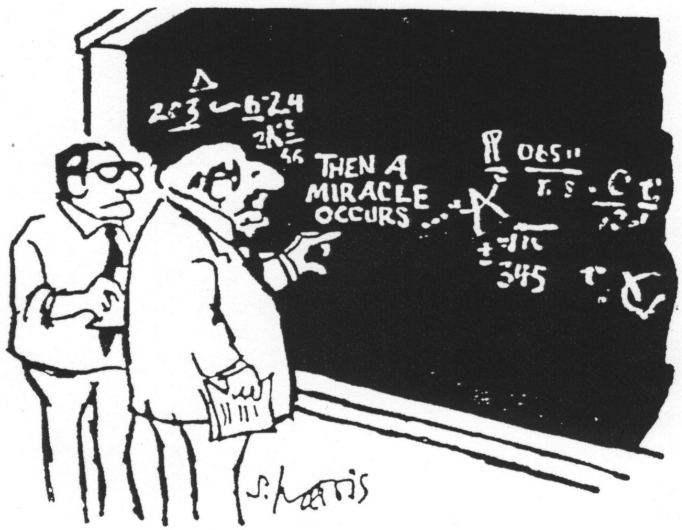
This is “invasive” profiling—change the source code of the program to add instrumentation (log statements).

Plus we have to do a lot of manual accounting.

Doesn’t really scale.

Wizardry!

Like debugging: if you get to be a wizard you can maybe do it by code inspection.



I think you should be a little more specific, here in Step 2

But speculative execution inside your head is harder for perf than debugging.

So: we want to use tools
and do this in a methodical way.

Introduction to Profiling

So far we've been looking at small problems.

Must **profile** to see what's slow in a large program.

Two main outputs:

- flat;
- call-graph.

Two main data gathering methods:

- statistical;
- instrumentation.

Profiler Outputs

Flat Profiler:

- Only computes average time in a particular function.
- Does not include other (useful) information (callees).

Call-graph Profiler:

- Computes call times.
- Reports frequency of function calls.
- Gives a call graph: who called what function?

Data Gathering Methods

Statistical:

Mostly, take samples of the system state, that is:

- every 100ms, check the system state.
- will cause some slowdown, but not much.

Instrumentation:
Add instructions at specified program points:

- can do this at compile time or run time (expensive);
- can instrument either manually or automatically;
- like conditional breakpoints.

Guide to Profiling

When writing large software projects:

- First, write clear and concise code.
  Don't do premature optimizations—
  focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Things to Look For

Good signs:

- Time is spent in the right part of the system.
- Most time should not be spent handling errors; in non-critical code; or in exceptional cases.
- Time is not unnecessarily spent in OS.

Kitchener driver follows GPS directions right into Lake Huron...

Development vs Prod

You can always profile your systems in development, but that might not help with complexities in production.

The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Per-Process

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.

Previous Offering Profiling

```
[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
```

Do it with Rust

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.

I’ve got a Cunning Plan

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.

CLion

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.