ECE459: Programming for Performance Winter 2023

Lecture 25 — Profiling: Observing Operations

Jeff Zarnett *2022-11-20*

Observations

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…

Why Observation? We're talking here about the idea of observation of our program, which is a little bit more inclusive than just measuring things, because we may observe things that are hard or impossible to quantify. Observing the behaviour of the program will obviously be super helpful in terms of figuring out what – if anything – to change. We have several different ways of looking at the situation, including logs, counters, profiling, and traces. Differences between them, briefly [Sit21]:

- **Counters** are, well, a stored count of the occurrences of something: how many times we had a cache miss, how many times we called `foo()`, how many times a user logged in...
- **Profiles** are higher-level overviews of what the program is doing, where time is going in aggregate, and how long certain operations take.
- **Traces** are recordings that show the sequence of events that the program is doing, so we can understand the runtime behaviour and how we got to the state we are in.

We'll return to profilers later, because it's a big topic, but for now we'll start with traces and counters.

[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.

Logging

We'll start with the idea of logging. It's an effective way of finding out what's going on in your program as it executes, and probably all of us have use print statements as a form of debugging, but also as a form of tracing. Logs typically have a timestamp, a message, and some attributes.

In [Sit21] it's recommended that if there can be only one tool for observing your program's execution, it is logging. This is because logging can tell you things about how much work the software did in a time period, when things are busy, which transactions are slow, when the service is down, etc. This probably matches well with our intuition that printf debugging or tracing is a good first option to figure out what's going on.

Given that we want to log, we do have to consider how often to log, and what should be in the content of it. A typical approach is to log an incoming request (input) and the outgoing response (output), and maybe some steps in between. We want to link the different things together (perhaps with attributes) so that we can see what happened. Here's a made-up example that doesn't show the time stamps or attributes:

```
Received request: update plan of company 12345 to ULTIMATE
Retrieving company 12345
Verifying eligibility for upgrade of company 12345 from BASIC to ULTIMATE
Company 12345 is not eligible for upgrade due to: unpaid invoices > 0
Returning response: update of company 12345 to ULTIMATE is DECLINED
```

We can see the various steps of the process and quickly understand what happened, because the logs show us the request, the stages of handling the request, and the response. In this example I intentionally added a decision and logged the reason for the decision. That's preferable to making someone guess why the answer was no.

Logging every request might be too noisy to actually find anything in the resulting data. The ability to search and sort helps, but it can still be going by too fast to realistically assess in real-time. Logs typically have levels, like error, warning, info, debug... These can make it easier to spot just the information that is relevant.

Typically, adding any relevant attributes is very helpful to identify what has happened or to correlate knowledge. If we can see in the attributes that updates for plans of companies always fail if the company's address is in Portugal, that's actually a useful observation.

With that said: there's such a thing as too much detail in logging. Logs should always redact personally-identifiable information such as people's names, contact information, medical records, etc. Unless the logs are really kept in secure storage with appropriate access controls, they can be seen by people who shouldn't have that information. Why should, for example, a developer on the website team have access to the banking details of a customer? My opinion: they shouldn't.

One way that we get into the situation of over-logging is if it's difficult or impossible to use other tools on the program or to have (read-)access to the database. If someone is trying to debug the behaviour of the program it might seem sensible to log just about everything because it's frustrating to try to debug a problem with incomplete information.

So we are aiming for a balance: include the relevant information that would let us spot patterns or correlate with other data sources, but not so much that the important information is lost or that PII is exposed.

References

[Sit21] Richard L. Sites. *Understanding Software Dynamics*. Addison-Wesley Professional, 2021.