ECE459: Programming for Performance Winter 2021

Lecture 20 — Self-Optimizing Software

Jeff Zarnett *2020-11-13*

Self-Optimizing Software

Our previous discussion about compiler optimizations (and optimizing the compiler) was focused on things that the compiler can do at compile-time. But what about runtime? The compiler can't do much at runtime, but a sufficiently-smart program can change itself at runtime to be better. Better, in this case, meaning faster. But what about change? We'll start with the simple things, and move on to the more complex and harder to get right. The simple stuff has to do with changing what's in memory, and well advance into changing the configuration, and finally, we'll consider changing the binary itself.

Caching. Your first thought about how a program might change itself for speed might be something like caching! Suppose that you keep track of the most popular exchange rates in memory, so that they are available faster than by going to the database. The management of the cache will be at runtime; the contents of the cache will be based on the actual usage. And, it will change over time to adapt to the patterns of usage: if today the exchange rate of CAD-EUR is popular, it will appear in the cache, and without any code changes if the exchange rate of CAD-GBP becomes popular, it goes in the cache and becomes faster to access. So, you *do* technically get different behaviour at runtime, but this is not quite what we wanted to talk about in this topic. It's too easy, and your program should probably be doing it already.

Observe and change. The next idea relates to having your program's configuration change at runtime to adapt to observed behaviour. If there are multiple routes to the same outcome, we might decide at runtime which is the best. This is effective because our initial guess might not be correct, but also because conditions can change at any time.

If you've taken a course in databases, you should have been exposed to the idea of query processing: given a certain query (`SELECT id FROM... JOIN... WHERE...`), the database server comes up with an execution plan for how to carry it out. For a simple query, there might be only one answer. A more complex query will have multiple correct answers and the database server will do what it considers best. However, that is based on an estimated cost only. The server could, at least once, try out a different strategy and notice if it is better than the original plan. A lot of queries happen many times (or are extremely similar to already-observed queries), so remembering what worked is helpful.

Building on that, the database server could change how it organizes the data based on what would be most efficient for the most common usage patterns. You can do the same in your program. For an analogy, I can sort the Excel sheet with student grades by either student ID (20xxxxxx) or user ID, based on whatever I use more. So if it is during the term and I'm entering grades, I probably use student ID as the way the file is sorted so I can get efficiently where I need to go. And I could always change that organization if it makes sense, such as after the end of term when people may e-mail me (including their user ID, but not student ID). The idea of re-structuring data storage can also apply to files on disk.

The observe-and-change strategy can also apply when invoking external services (external as in, over the internet). Suppose there are three different servers where we can send messages and we'll measure and remember how long it took to get a response from that server. The fastest server might be the one that is the closest geographically, but that server might be very busy, so it might be faster to communicate with a server that's less busy but farther away. Maybe you send 8 out of every 10 messages to the server that was most recently determined to be the fastest, and one to each of the other two servers. You might discover that your current guess at the fastest server is wrong and it's better to switch your primary.

Genetic algorithms. If you've taken an advanced course on algorithms, you might have covered genetic algorithms in some detail. This isn't going to be a replacement for that, but will just give you an idea of how the idea can be used. First, a quick three-paragraph explainer on genetic algorithms [Whi98].

A genetic algorithm is inspired by the idea of natural selection. Our program is trying to solve a particular problem. A number of candidate solutions are created (usually, randomly) and they are evaluated for their fitness: how well do they solve the problem? Solutions with a higher fitness have a higher chance of continuing forward into the next group of candidates, called the next generation. At each generation, good solutions from the previous generation are combined (if possible) and/or mutated randomly to see if that makes the solution better. This process repeats until a sufficiently-optimal solution is found, or a fixed number of generations have been evaluated. Thus, solutions with good qualities "reproduce" and move forward in the simulation, and those with bad qualities "die out" and we do not continue down that path. If we do this well, eventually we end up with a solution that's good, or at least good enough.

This works for the kind of problem where we, first of all, have some parameters to configure. If there is nothing to configure, there's nothing to change or evolve. We also need a fitness function that allows us to evaluate how well the problem is being solved. This function cannot be binary (pass/fail) because that doesn't show whether a given solution A is better or worse than solution B; instead we want a continuous (in the mathematical sense) definition of fitness so we can say solution A with 84.1 is better than B with 81.0.

Of course, a genetic algorithm does not necessarily guarantee the best possible outcome. It is possible that the fitness function tends towards a local-maximum rather than the global one, so we get a good solution but not the best. Similarly, the fitness of a solution might be evaluated rarely or take a long time, making the process of finding a good solution slow.

Right, with that in mind, you might ask how genetic algorithms help in making your program faster. The typical use for a genetic algorithm is something like designing an antenna or an airplane wing where making random changes gives some numbers. We can do the same with a generic program, if ti has the right properties and what we're trying to do is optimize our configuration parameters.

Let's return to the subject of Google Maps. In our earlier discussion on early-phase termination, we tried to brainstorm ideas about how potential routes are generated and how we know if we have enough or a good-enough route. In the discussion, we will imagine that the decision of when to terminate the search is based on when we have a "good enough" solution. Then there are the various parameters that go into generating a solution, which I'll guess to be something like:

- Number of routes to evaluate
- Heuristic for generating routes to evaluate
- Decay of traffic information reported by other motorists
- Time of day and month and if it's a holiday
- Search radius for alternate routes

It is Google, after all, so they probably consider many more parameters or use a completely different mechanism. It might be very difficult to choose what the correct values for these parameters are (especially if they vary by time of day, day of the week, on holidays, etc). Changing them by hand probably does not work, but letting a genetic algorithm choose the values based on experimenting and trading off the quality of the solution against the time to come up with it. We might consider a solution that only comes up with awful routes to fail, even if it gets them nearly instantly. And we might consider the successful solution one that comes up with a route that is optimal or nearly-optimal in the shortest time.

One reason why genetic algorithms might be a good choice for this kind of problem is that the problem is nonlinear: that is, we cannot treat each parameter as an independent variable and change just one and expect that the change in output is only a result of the change of the one input variable and not also an interaction of one variable with others [Whi98].

Perhaps this takes away some of the mystery of genetic algorithms. Maybe you're thinking this isn't really self-optimizing software, it's just optimizing configuration parameters. Let's go up (well, really, down) a level, then. Now we'll move into changing the binary itself.

Hotspot. The previous discussion of compiler optimizations talked about all the things that compiler can do to make the program more efficient. Some of them are always a clear win. Precomputing something or skipping unnecessary code is always going to be better than the alternative. Other optimizations are not. Let's consider the decision about inlining: sometimes it's good, but sometimes it doesn't help or makes things worse. In those cases, the compiler has to take a decision about whether to do it or not and that's what is in the binary.

In JVM-languages like Java, the virtual machine itself can do some optimization because of the just-in-time (JIT) compiler. The original program is turned into the intermediate representation by the Java compiler, and then there's a second chance at runtime! Oracle's documentation tell us that there's actually two different JIT compilers; one for clients and one for servers. The client one produces its output faster, but that code produced will be less efficient. The server one takes more time and more resources to produce slightly better code.

The major advantage that the JIT compiler has is being able to observe the runtime behaviour of the program and then change its decision. If we see that, for example, inlining would be helpful but the original decision was not to do it, we can change that decision. This is very helpful in scenarios like inlining, because we'll have the function call overhead every single time and therefore every call to the function increases the penalty of getting the decision wrong. Being able to change our decision is less helpful when it comes to something like a branch prediction, because the hardware will most likely save us if our prediction isn't very good (see our earlier discussion on this topic).

There are actually a few other things that can be done by the JIT compiler at runtime which are not likely to be doable at compile time. In particular, I want to focus on *escape analysis on-stack replacement*, and *intrinsics* as outlined in [Str18].

Let's start with escape analysis. The purpose of this is to figure out if there are any side effects visible outside of a particular method. This allows some skipping of heap allocation and we can stack allocate them instead. The more interesting thing are possible lock optimizations: lock elision, lock coarsening, and nested locking. Lock elision: if the JIT compiler can determine that a lock serves no purpose – e.g., a method or block is tagged as `synchronized` but it can only ever get called from one thread at a time – there's no lock at all and therefore no setup an1d acquisition costs. Lock coarsening: if there are sequential blocks sharing the same lock, the compiler can combine them to reduce the amount of locking and unlocking that needs to happen. The cuts down the overhead and we'll take it where we can. Nested locks: if the same lock is required repeatedly without releasing (in some recursive code, perhaps) this can also be combined so we don't have as much overhead lost to locking and unlocking.

On-stack replacement is a way that the virtual machine can switch between implementations of a particular method. This is helpful if a function is identified as important (sometimes called *hot*) in that it runs frequently, then a more optimized version can be swapped in. This does have a slight cost, in that the virtual machine has to pause execution briefly to swap in the new stack frame (as it may be organized a differently), but it will be of some benefit in the long run if this function is truly a frequently-executed piece of code.

Intrinsics are highly-optimized versions of code that is precompiled for a specific platform (e.g., x86). If a particular piece of code is truly critical, then using that native implementation might be faster, but it might not always be available, depending on your platform. These were originally developed in C++ rather than Java so it's also done without the safety rails of Java. I guess in that sense you could say that C++ is Java's unsafe mode. That might be a controversial statement. Now there's a different approach called Graal, which turns the Java bytecode directly into machine code. Adding an intrinsic is a complicated process

The hotspot approach is probably closest to what you might have imagined by reading the title of this topic: changing the binary code based on observed runtime behaviour of the program. Rust doesn't have the capability of detecting the parameters at runtime and swapping binary code them, because it has chosen not to have a runtime like the JVM. While that does mean that there's less overhead in general, the tradeoff is that any decisions made at compile time will remain so. Unless...

Rewriting the binary.

References

[Str18] Jakub Stransky. Hotspot jvm jit optimisation techniques, 2018. Online; accessed 2020-11-13. URL: https://jakubstransky.com/2018/08/28/hotspot-jvm-jit-optimisation-techniques/.

[Whi98] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4, 10 1998. doi:10.1007/BF00175354.