Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Yet Another Optimization Article

Martin Fowler

This is a troubling column to write. I hadn't planned to write on optimization, because what I have to say has already been said numerous times. Yet, when I give talks, I find there's still a surprising number of people who don't know, or at least don't follow, the advice I'm about to give. So, here goes. (Many of you have probably seen this advice before—my thought to you is to ponder why I need to say this again.)

First, performance matters. Although relying on Moore's law to get us out of slow programs has its merits, I find it increasingly annoying when I get a new version of a program and must upgrade my hardware for it to work acceptably. The question is, "How do we achieve a fast program?"

For many programmers, performance is something you pay continuous attention to as you program. Every time you write a fragment of code, you consider the performance implications and code the program to maximize performance. This is an obvious technique—pity it works so badly.

Performance is not something you can work on in this way. It involves specific discipline. Some performance work comes from architectural decisions, some from a more tactical optimization activity. But what both share is the fact that it is difficult to make decisions about performance from just looking at the design. Rather, you have to actually run the code and measure performance.

Steps for optimization

Optimizing an existing program follows a specific set of steps. First, you need a *profiler*—a program that can analyze how much time your program spends in its various parts. Software performance has an 80/20 rule: 80 percent of the program's time is spent on about 20 percent of the code. Trying to optimize performance in the 80 percent of code is futile, so the first order of business is to find that 20 percent of code. Trying to deduce where the program will spend its time is also futile. I know plenty of experienced programmers who always get this wrong. You have to use a profiler.

To give the profiler something to chew on, perform some kind of automated run that reasonably simulates the program under its usual conditions. An automated test suite is a good starting point, but make sure you simulate the actual conditions. My colleague Dave Rice has a rule: "Never optimize a multiuser system with single-user tests." Experience has taught us that a multiuser database system has very different bottlenecks than a single user system—often focused around transaction interactions. The wrong set of tests can easily lead you to the wrong 20 percent of code.

Once you've found your bottlenecks, you have two choices: speed up the slow things or do the slow things less often. In either case, you must change the software. This is where having a well-designed piece of software really helps. It's much easier to optimize cohesive, loosely coupled modules. Breaking down a system into many

0740-7459/02/$17.00 © 2002 IEEE

small pieces lets you narrow down the bottlenecks. Having a good automated test suite makes it easier to spot bugs that might slip in during optimization.

It's worth knowing about various optimization tricks, many of which are particular to specific languages and environments. The most important thing to remember is that the tricks are not guaranteed to work—as the saying goes, "Measure twice, cut once." Unlike a tailor, however, you measure before and after you've applied the optimization. Only then do you know if it's had any effect. It's revealing how often an optimization has little—or even a negative—effect. If you make an optimization and don't measure to confirm the performance increase, all you know for certain is that you've made your code harder to read.

This double measurement is all the more important these days. With optimizing compilers and smart virtual machines, many of the standard optimizing techniques are not just ineffective but also counterintuitive. Craig Larman really brought this home when he told me about some comments he received after a talk at JavaOne about optimization in Java. One builder of an optimizing virtual machine said, in effect, "The comments about thread pools were good, but you shouldn't use object pools because they will slow down our VM." Then another VM builder said, "The comments about object pools were good, but you shouldn't use thread pools because they slow down our VM." Not only does this reinforce the need to measure with every optimization change, it also suggests that you should log every change made for optimization (a comment tag in the source code is a good option) and retest your optimizations after upgrading your compiler or VM. The optimization you did six months ago could be your bottleneck today.

All of this reinforces the key rule that first you need to make you program clear, well factored, and nicely modular. Only when you've done that should you optimize.

Some exceptions

Although most performance issues are best dealt with in these kinds of optimizations, at times other forms of thinking are important—for example, during early architectural stages, when you're making decisions that will be costly to reverse later. Again, the only way to really understand these performance issues is to use measurements. In this case, you build exploratory prototypes to perform crude simulations of the environments with which you're going to work and to get a sense of the relative speeds. It's tricky, of course, to get a good idea of what the actual environment might be, but then it's likely that everything you're working with will be upgraded before you go live anyway. Experiments are still much better than wild guesses.

There are also some cases where there are broad rules about slow things. An example I always come across is the slowness of remote procedure calls. Because remote calls are orders of magnitude slower than in-process calls, it's important to minimize them, which greatly affects overall design. However, this doesn't trump measuring. I once came across a situation where people optimize remote methods only to find their bottlenecks were elsewhere. However, minimizing remote calls has proven to be a solid rule of thumb.

If you make an optimization and don't measure to confirm the performance increase, all you know for certain is that you've made your code harder to read.

Some have taken this further, coming up with performance models that you can use to assess different architectural designs. Although this can be handy, it's difficult to take it too far. It all depends on how good the performance model is, and people usually cannot predict the key factors, even at a broad level. Again, the only real arbiter is measurement.

In the end, however, performance is not an absolute. Getting a program to run faster costs money, and it's a business decision whether to invest in a quicker program. One value of an explicit optimization phase is that the cost of getting fast performance is more explicit, so businesses can trade it against time to market or more features. Much of the reason why I curse at slow software is because it makes good business sense for the builder.

As I've said, much of this advice—in particular, the advice to write a good clean program first and optimize it later—is well worn. (For a longer description of this approach, read Chapters 28 and 29 of Steve McConnell's *Code Complete*, Microsoft Press, 1993.) Good quotes about the perils of premature optimization have been around for over 20 years. The pity is that some people still object when I call the same query method twice in a routine. For everyone who finds nothing new in this column, there exists another challenge—how to make it so there is no need to rewrite it in 10 years.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.