You work in the product development group of a software company, where the product is often compared with the competition on performance grounds. Performance is an important part of your business; but so is adding new functionality, fixing bugs, and working on new projects. So how do you lead your team to develop high-performance software, as well as doing everything else? And how do you keep that performance high throughout cycles of maintenance and enhancement?

TELL YOUR PEOPLE

If performance is important to your business, your employees need to know. Throughout your development team, you're looking for a balanced performance culture—not the kind where developers spend hours profiling and optimizing deep into the night, missing deadlines as a result, but the kind where people feel within their rights to comment that a particular feature is a bit slow. Developers need to believe that, fundamentally, their product is high-performance, and that something is wrong if it's not.

Unfortunately, many organizations isolate their developers from commercial pressures; they don't want their technical people spending valuable development time on presales work, or they don't want to disclose sensitive

From design to production,
performance should be
part of the process.

High-Performance

Team

PHILIP BEEVERS, ROYALBLUE

High-Performance Team

information to a large number of employees. Presales information on performance constraints or comparisons, however, is hugely valuable, and development teams need to be made aware of it; little energizes them more than the realization that a customer is considering an alternative solution because it is quicker on a given benchmark. Tell your developers when a customer signs up on the basis of product performance; tell them when your product is being benchmarked against a rival; and don't be afraid to tell them when your product is slower than a competitor's.

What you're trying to do is ingrain the idea of high performance at a subconscious level. You quite clearly don't want your developers thinking that performance is the most important thing; for almost all applications, it isn't. Unless everyone has some awareness of the importance of performance, however, you can expect a tail-off as your application is maintained. Successive releases will require more memory, start slightly slower, or be just a little less responsive. Although this isn't a problem for your new customers, who are probably buying the latest, fastest hardware to run your application, it causes pain to your existing customer base—it may be that a new release will no longer fit on the existing hardware platform, or a critical business operation suddenly takes a little bit too long.

THE RIGHT DESIGN

Designing for performance is a controversial area; there are those who think you must always start by designing for performance, and others who think you should start with something that works and optimize it later. Both approaches have their merits; as always, it's a case of finding the right balance between the two.

High-level design decisions are often hard to change, and thus are fundamentally tough to optimize. Therefore, at this level, you must consider performance—interfaces between major components, public APIs, and database schemas all fall into this category—particularly as modifications make upgrades difficult. Lower-level design points—for example, a private, nonpersistent data structure—are easier to change, so it's best to start with something easy to understand and optimize it when it proves to be a problem.

At this point, you need to remember that you've already told everyone that performance is important, so they can be trusted to implement those low-level details

with performance in mind. Your job now is to encourage experimentation: Rather than theorizing, ask developers to hack together 50-line test rigs to contrast different approaches to the same problem. If a particular algorithm or data structure has been chosen on the basis of performance or efficiency, ask to see the evidence—and say why. You're not trying to prove anyone wrong or make them look silly; you just want to know they have thought about it and can justify their decisions. What's more, you want people to back up those thoughts with experimental evidence; you don't want decisions made based on experience or prejudices gained on an old platform or in a previous job.

When reviewing some code, I once saw a potentially serious performance problem that could have been avoided by this experimental approach. Short, string-valued keys were used in a hash table, where the hashing algorithm had been written in such a way that it could return only one of 4,096 possible values. This was particularly odd because the programmer had sized the table for several tens of thousands of items; he was clearly expecting many more than 4,096 distinct key values, which made the choice of algorithm seem strange, and potentially performance sapping. Not only that, but the hash function seemed more complex, and thus slower, than our standard string-hashing algorithm taken from a textbook.

When I asked the programmer about this, he said, "It seems fine. I've used that hash algorithm in loads of places."

"So you didn't think of using our standard one?" I continued.

"No, this one is better."

"Why?"

"Oh, I can't remember. I think I tested them once and found a case where mine was better."

When I ran some tests of my own, I showed that for realistic workloads, the standard algorithm was faster and always generated a wider spread of keys.

CODE REVIEWS

This example brings us to code reviews. Peer review of code is good practice, and I'm sure you're doing it, anyway; but are you considering performance when you do so? Are your reviewers looking for potential problems in performance-critical areas? Again, you need to strike a balance: Not everything needs to be high performance,

and there's no point optimizing something before it can be seen to be a problem; but a reviewer might spot a data structure that could become inefficient on certain classes of production data, or might be able to target performance testing.

Remember, there are only two ways to make software go faster: Make it do less stuff; and do what you're doing quicker.

Most people—particularly less experienced staff—will dive in and have a go at the latter, because it's a bigger intellectual challenge, and it's the programmer's mentality to break a problem down into small, manageable chunks. The big wins, however, are often there to be had with the former; you can't make `strcmp(3C)` any faster, so how about just calling it less often?

An example of this is a piece of code I've maintained for several years that is used to choose a relevant index to satisfy a database query, based on the fields specified by the user. This particular query engine can accept multiple sub-queries, logically ORed together to form a larger criterion. Historically it has worked by splitting the query into sub-queries and then choosing an index for each one. This is simple, but potentially not optimal—it can get slow for a large number of sub-queries.

For a while, I kept chipping away, incrementally speeding up the code that chose an index for each sub-query. From release to release, I'd be chipping off 10 or 20 percent, significant when you add it all together. The best optimization I could possibly make, however, was to run this code less—I found that in many cases, the sub-queries ended up choosing the same index anyway. If I could group the similar sub-queries together up front, I could massively reduce the number of times I had to call the index choice code. This wasn't a 10 or 20 percent improvement; this change increased the performance of the best case by 100 times!

A corollary to this is that you need to look hard at the common-case usage of your application, at the possible expense of other cases. By definition, this is the code path that is executed the most, so optimizing it has the most impact on overall performance.

The biggest performance improvements are almost always at the highest levels of the software stack, so when you're optimizing a system, think about the architecture of the performance-critical pieces; break it down into steps and ask: Do we really need to do each of these steps? Do we really need to do each of these steps for the most common case?

Although this article isn't about optimizing software per se, as a senior engineer or product manager, you need

to be thinking about these areas when assigning resources to performance work. Where junior staffers are involved, be sure you have directed them to look at the right pieces of the problem.

DON'T GUESS. USE THE TOOLS

Some engineers have great intuition when it comes to performance. They can look at a problem, glance at a few favorite statistical runs, and predict exactly how it should be fixed.

In my experience, we're talking about a very, very small percentage of engineers with this type of intuition. The vast majority of us mere mortals have dreadful intuition: We're almost always wrong and can waste a huge amount of time guessing, and guessing wrong; worse still, we introduce unnecessary risk by optimizing the wrong parts of our systems. We should be relying on the tools of our trade and developing a systematic approach to solving problems.

Remember, there are only two ways to make software go faster: make it do **less stuff; and do what you're doing **quicker**.**

Those of us with an interest in performance should be familiar with the profiling tools on our platforms of choice. Many commercial compiler suites include some sort of profiler (for example, the performance analyzer in the Sun Studio Compiler Collection), and other stand-alone products are available (such as Rational Quantify and Intel VTune). No list of useful tools would be complete without DTrace, the dynamic tracing framework in Solaris that is unique in its ability to look at systemic performance problems, examining the entire environment rather than a particular process or piece of code.

You undoubtedly have your favorite tools, so rather than recommend my own, the key message is to know those tools and apply them systematically. When you're optimizing, repeat the same runs and measure performance improvements using objective measures taken from those tools.

TEST

If you want to keep producing high-performance software, you *must* be able to run reproducible, comparable performance tests. Ideally, you'll have dedicated, standard hardware on which to run these tests; this should be representative of, if not directly comparable with, what your customers run in production. You'll run a basic set of performance tests as part of your release cycle, plus more comprehensive benchmarks as required.

So what should you test? What is important? You need to find a balance between the time it takes to run the tests and the information they actually give you. A large set of complex tests can tell you a huge amount about your application and even help you track down areas that have caused performance degradation, but that might be too time consuming to run for every release. Simpler tests that can run automatically in less than an hour would be better. Furthermore, your tests need to measure something using public interfaces that are stable between releases; otherwise, maintaining the tests will become an overhead.

Of course, the tests must exercise the operations and code paths that are important to your customers. They must measure the throughput of the common transactions or queries, based on the types of datasets and loadings seen on production systems. If practical, a captured production workload that can be rerun on demand would be ideal.

PUBLISH BENCHMARKS

Publishing benchmarks often seems like making a rod for your own back. Your customers read them, and that's the performance they expect. That's true to some extent—benchmarks can easily be misinterpreted—but customers interested in performance are sensible enough to realize that benchmarks aren't necessarily an indication of the absolute real-world performance they should expect; however, they are useful indicators of *relative* performance.

Publishing benchmark results is one of your biggest weapons in spreading the performance gospel within your own organization. It shows everyone working for the company the sort of performance the product can achieve—and it shows you're serious about measuring and improving that performance. Best of all, results start the discussion with your customers. They will soon tell you if your benchmark results are poor, or if they're

run on unrealistic hardware, or if the workloads are inappropriate.

Ideally, the benchmark results you publish will be the output of the tests you run at release time. If not, you're going to need to commit resources to keeping published benchmarks up to date.

TAKE AN INTEREST IN PRODUCTION SYSTEMS

You need to engineer for the cases that are common for your customers; your benchmarks and release tests need to be as representative as possible of real workloads; your design and implementation decisions need to strike the right compromise for production datasets. For each of these, it's critical that you have a familiarity with production systems. You need to see your software in the field, observe where it is stretched, and draw conclusions based on behavior across the customer base.

On top of this, you need to know what hardware your customers are using, and how it copes with their workload. You need to familiarize yourself with the physiology of a healthy system and view some real-world problems, so you can see where the warning signs start to appear. You need a good "feel" for real systems, and whether they're CPU-, memory-, disk-, or network-bound, and what business operations are performance-critical. You need to be familiar with the different kinds of workloads out there, and whether they require different tuning or configuration settings.

The importance of understanding your customers' production systems cannot be understated. What's more, you need the experience of actually gaining that understanding. It should provide insight into how monitoring tools and instrumentation can help you find a problem. Not everyone in the organization needs this understanding, but *someone* does, and you need to harness that understanding when it comes to designing future releases and benchmarks. At the very highest level, it's this understanding that drives architectural change for performance's sake.

PROVIDE INSTRUMENTATION

To achieve that understanding of production systems, you're going to need some instrumentation in your application. At some level, you need to know what the system is up to: what types of operations it is doing, how long they're taking, etc. Obviously, this instrumentation needs to be lightweight—the overhead must be negligible—

and the ideal is for crucial statistics to be enabled and recorded automatically on every system. This allows you to look at a system after a particular event—for example, a very busy day or a problem reported by users—and view those statistics, rather than having to enable them and wait for it to happen again.

Again, knowing the patterns of a healthy system can help you to spot problems quickly via your statistics. For example, if you log the number of transactions executed and the reads on each database table, you can quickly spot when a particular transaction is reading a given table too much. This might be the clue you need to track down a particular issue.

PUBLICIZE PROBLEMS

The software industry is wildly enthusiastic about the concept of re-use—re-use my script, re-use my code through an API, re-use my design methodology. What we don't seem to be quite so good at sharing is the pathology of problems—and this is particularly true of performance problems.

Within your organization, within the applications you write, each of your engineers is gaining experience in what's quick and what's not within the technologies you use. Those who care about and look closely at performance know which are the fastest, most memory-efficient data structures; they know the types of data structures that always cause problems; they understand the foibles of the libraries you use and how to deal with them. Sharing this technical litany is difficult; doing it well can derail less experienced staff. Again, code reviews play a role. Your more experienced engineers will be able to feed back what they don't like the look of and suggest better alternatives.

Performance problems on production systems also need publicizing, so your whole organization can think about how to avoid them next time. Maybe the customer was doing something a little unexpected—but why? Do you need to document a better alternative so the customer doesn't do it in the future? Or is it a sensible business practice that you need to support better?

Performance is a specialist area; if it's important to your business, you may need performance specialists on your development and support teams who can use their experience to address issues quickly. These people are only the spearhead, however. If performance is important, the entire development team needs to be aware of it and involved in it. One or two highly skilled performance experts can point you in the right direction when problems arise, but they can't optimize every piece of code.

PERSEVERANCE

Performance isn't everyone's top priority; as already mentioned, for most applications performance is at best one of a large number of competing and potentially conflicting requirements. With limited resources, it's always difficult to prioritize, and something has to give; if there isn't an immediate commercial pressure to make the application fast, the slipping point may be performance.

You just need to keep trying: Continue to suggest improvements and work to keep performance on everyone's radar screen. Perhaps most importantly, you need to be strategic. Think about how your product could be changed to avoid the problems you've seen in the field or to radically increase performance. Perhaps this might consist of a technical change—a move to 64-bit or a grid paradigm—or some sort of re-organization of business logic, to cut out unnecessary processing for common cases. Identify where the bottlenecks are, and think what you would do with a clean sheet of paper to eliminate them; then look at how you can incorporate the changes into your current designs or perhaps migrate in the chosen direction.

IT'S ALL ABOUT BALANCE

Finally, as ever in software, performance is all about balance. There's a trade-off between how quickly the application runs and how much effort you put into optimizing it—and thus, indirectly, how much effort you put into implementing other features. Your organization needs to find the right compromise and prioritize performance accordingly.

Developing high-performance software is hard; in addition to being technically difficult, it offers managerial and operational challenges. Organizations need to subtly change the way they market, design, build, and support their products to keep producing software that meets the customers' performance requirements. ☐

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

PHILIP BEEVERS manages the infrastructure development team at royalblue, a software company that provides market-leading, high-performance financial trading systems. He has been involved in this application area for nine years, working on a proprietary 64-bit in-memory database, low-latency middleware and event notification, and more general performance analysis. He graduated from Oxford University with a B.A. in mathematics.

© 2006 ACM 1542-7730/06/0200 \$5.00