12

Optimization

Premature optimization is the root of all evil.

— C. A. R. Hoare

This is going to be a very short chapter, because the main thing Unix experience teaches us about optimizing for performance is how to know when not to do it. A secondary lesson is that the most effective optimization tactics are usually things we do for other reasons, such as cleanness of design.

12.1 Don't Just Do Something, Stand There!

The most powerful optimization technique in any programmer's toolbox is to do nothing.

This very Zen advice is true for several reasons. One is the exponential effect of Moore's Law—the smartest, cheapest, and often *fastest* way to collect performance gains is to wait a few months for your target hardware to become more capable. Given the cost ratio between hardware and programmer time, there are almost always better things to do with your time than to optimize a working system.

We can get mathematically specific about this. It is almost never worth doing optimizations that reduce resource use by merely a constant factor; it's smarter to concentrate effort on cases in which you can reduce average-case running time or space use from $O(n^2)$ to $O(n)$ or $O(n \log n)$[1], or similarly reduce from a higher order. Linear performance gains tend to be rapidly swamped by Moore's Law[2].

Another very constructive form of doing nothing is to not write code. The program can't be slowed down by code that isn't there. It can be slowed down by code that *is* there but less efficient than it could be—but that's a different matter.

12.2 Measure before Optimizing

When you have real-world evidence that your application is too slow, then (and *only* then) is the time to think about optimizing the code. But before you do more than think about optimizing, measure.

Recall Rob Pike's six rules in Chapter 1. One of the lessons that the original Unix programmers learned early is that intuition is a poor guide to where the bottlenecks are, even for one who knows the code in question intimately. Unixes, unlike most other operating systems, usually come with profilers; use them.

Reading profiler results is something of an art. There are a couple of recurring problems: one is instrumentation noise, another is the effect of imposed external latencies, and a third is overweighting of upper nodes in the call graph.

The instrumentation-noise problem is fundamental. Profilers work by inserting instructions that report execution time at the entry and exit points of subroutines, also at fixed intervals within the inline code of routines. These instructions themselves take time to execute. The effect is to reduce the dispersion of call times: very short subroutines tend to look more expensive than they are, with a lot of noise in their comparative call times, while for longer ones the instrumentation overhead is invisible.

---

1. For readers unfamiliar with O notation, it is a way of indicating how the average running time of an algorithm changes with the size of its inputs. An $O(1)$ algorithm runs in constant time. An $O(n)$ algorithm runs in a time that is predicted by `An + C`, where `A` is some unknown constant of proportionality and `C` is an unknown constant representing setup time. Linear search of a list for a specified value is $O(n)$. An $O(n^2)$ algorithm runs in time An^2 plus lower-order terms (which might be linear, or logarithmic, of any other function lower than a quadratic). Checking a list for duplicate values (by the naïve method, not sorting it) is $O(n^2)$. Similarly, $O(n^3)$ algorithms have an average run time predicted by the cube of problem size; these tend to be too slow for practical use. $O(\log n)$ is typical of tree searches. Intelligent choice of algorithm can often reduce running time from $O(n^2)$ to $O(\log n)$. Sometimes when we are interested in predicting an algorithm's memory utilization, we may notice that it varies as $O(1)$ or $O(n)$ or $O(n^2)$; in general, algorithms with $O(n^2)$ or higher memory utilization are not practical either.

2. The eighteen-month doubling time usually quoted for Moore's Law implies that you can collect a 26% performance gain just by buying new hardware in six months.

Bearing instrumentation noise in mind, it's wise to assume that the times listed for the fastest, shortest subroutines are going to have a lot of froth and air in them. They can still be eating a lot of time if they are called very frequently, however, so pay particular attention to their call-count statistics.

The external-latency problem is also fundamental. There are various sorts of delay and distortion that can happen behind the profiler's back. The simplest is overhead from operations with unpredictable latency—disk and network accesses, cache fills, process-context switches, and the like. The problem is not so much that these overheads happen—they may actually be what you're trying to measure, especially if you're focusing on whole-system performance rather than just tuning a critical inner loop. The problem is that they have a random component that means the results from any individual profiling run may not be very useful.

One way to minimize the effects of these noise sources, and get a better picture of where the time is going in the average case, is to add together the results from a lot of profiling runs. There are a lot of good reasons to build test harnesses and test loads for your programs before you get to optimizing; the most important reason, usually far more important than performance tuning, is so you can regression-test your program for correctness as you change it. Once you've done this, being able to profile repeated tests under load is a nice side effect that will often give you better information than a few runs by hand.

Various effects tend to allocate time spent to calling routines rather than callees, overweighting upper modes in the call graph. Function-call overhead, for example, is often charged to the calling routine (whether or not this is true depends partly on your machine architecture and where the profiler is allowed to insert probes). Macros and inline functions, if your compiler supports them, won't show up in the profiling report at all; every bit of their time gets charged to the calling function.

More importantly, many time-reporting tools give a display in which time spent in subroutines is charged to the caller. (The *gprof*(1) profiler distributed with open-source Unixes has this trait.) Naïvely subtracting callee time from caller time won't give you a useful result if the same routine can have more than one caller—the effect would be to artificially deflate both callers' times. Especially nasty is the common case of a utility function with multiple call sites, some of which make lots of trivial calls and others of which make a few complicated ones.

To get more transparent results, factor your code so that upper-level routines consist as much as possible of calls to lower-level routines, rather than in-line code. If you keep the overhead of upper-level control logic to a minimum, the call structure of the code will tend to organize the profile report in a way that is relatively easy to read.

You'll get more insight from using profilers if you think of them less as ways to collect individual performance numbers, and more as ways to learn how performance varies as a function of interesting parameters (e.g., problem size, CPU speed, disc speed, memory size, compiler optimization, or whatever else is relevant). Try fitting those numbers to a model, using open-source software like R or a good-quality proprietary tool like MATLAB.

> The natural smoothing of the data that results from model fitting tends to focus on the big effects and cover up the small, noisy ones. For example, by fitting a cubic to the matrix inversion routine in MATLAB on random matrices from 10×10 to 1000×1000, it is clear that we actually have three cubics, with clearly defined boundaries, that correspond roughly to "in cache", "in memory but out of cache", and "out of memory". The data shows us this effect even if weren't looking for it, just by looking at the deviations from the best fit.
>
> —Steve Johnson

12.3 Nonlocality Considered Harmful

The most effective way to optimize your code is to keep it small and simple. We've been through lots of good reasons to keep it small and simple earlier in this book. Here's a new one: you want the central data structures and the time-critical loops in your code never to fall out of cache.

Consider your target machine as a hierarchy of memory types arranged by distance from the processor. There are the processor's own registers; its instruction pipeline; the level-one (L1) cache; the level-two (L2) cache; possibly a level-three (L3) cache; main memory (what Unix old hands still quaintly call 'core'); and the disk drives where swap space lives. Technologies like SMP, shared-memory clusters, and non-uniform memory access (NUMA) add more layers to the picture but only widen the overall spread.

Every kind of access to that stack is getting faster. Processor cycles are almost free, outside of a few demanding applications like modeling nuclear explosions or real-time video compression. But what's also happening is that the speed ratios between layers in the storage hierarchy are all increasing as processor speeds go up. Thus, the relative cost of a cache miss is increasing.

So we have an interesting paradox. As machine resources plummet, the expected cost of large data structures falls—but because the cost spread between adjacent cache levels is also going up, the performance impact of being just large enough to break a cache boundary is also rising.

"Small is beautiful" is therefore better advice than ever, particularly with regard to central data structures that must live in the fastest possible cache. The advice applies to code as well; the average instruction spends more time being loaded than it does executing.

This turns some traditional advice on its head. Compiler optimizations like loop unrolling, which get rid of relatively expensive machine instructions in return for an increase in total code size, may no longer be worth doing. Another example is precomputing small tables—for example, a table of sin(x) by degree for optimizing rotations in a 3D graphics engine will take 365 × 4 bytes on a modern machine. Before processors got enough faster than memory to demand caching, this was an obvious speed optimization. Nowadays it may be faster to recompute each time rather than pay for the percentage of additional cache misses caused by the table.

But in the future, this might turn around again as caches grow larger. More generally, many optimizations are temporary and can easily turn into pessimizations as cost ratios change. The only way to know is to measure and see.

12.4 Throughput vs. Latency

Another effect of fast processors is that performance is usually bounded by the cost of I/O and—especially with programs that use the Internet—network transactions. It's therefore valuable to know how to design network protocols for good performance.

The most important issue is avoiding protocol round trips as much as possible. Every protocol transaction that requires a handshake turns any latency in the connection into a potentially serious slowdown. Avoiding such handshakes is not specifically a Unix-tradition practice, but it's one that needs mention here because so many protocol designs lose huge amounts of performance to them.

> I cannot say enough about latency. X11 went well beyond X10 in avoiding round trip requests: the Render extension goes even further. X (and these days, HTTP/1.1) is a streaming protocol. For example, on my laptop, I can execute over 4 *million* 1x1 rectangle requests (8 million no-op requests) per second. But round trips are hundreds or thousands of times more expensive. Anytime you can get a client to do something without having to contact the server, you have a tremendous win.
>
> —Jim Gettys

In fact, a good rule of thumb is to design for the lowest possible latency and ignore bandwidth costs until your profiling tells you otherwise. Bandwidth problems can be solved later in development by tricks like compressing a protocol stream on the fly; but getting rid of high latency baked into an existing design is much, much harder (often, effectively impossible).

While this effect shows up most clearly in network protocol design, throughput vs. latency tradeoffs are a much more general phenomenon. In writing applications, you will sometimes face a choice between doing an expensive computation once in anticipation that it will be used several times, or computing only when actually needed (even if that means you will often recompute results). In most cases where you face a tradeoff like this, the right thing to do is bias toward low latency. That is, don't try to pre-compute expensive operations unless you have a throughput requirement and know by actual measurement that the throughput you are getting is too low. Pre-computation may seem efficient because it minimizes total use of processor cycles, but processor cycles are cheap. Unless you are doing one of a handful of monstrously compute-intensive applications like data mining, animation rendering, or the aforementioned bomb simulations, it is usually better to opt for short startup times and quick response.

In Unix's early days this advice might have been considered heretical. Processors were much slower and cost ratios were very different then; also, the pattern of Unix use was tilted rather more strongly toward server operations. The point about the value of low latency needs to be made partly because even newer Unix developers sometimes inherit an old-time cultural prejudice toward optimizing for throughput. But times have changed.

Three general strategies for reducing latency are (a) batching transactions that can share startup costs, (b) allowing transactions to overlap, and (c) caching.

12.4.1 Batching Operations

Graphics APIs are frequently written on the assumption that the fixed setup cost for a physical screen update is large. Consequently, the write operations actually modify an internal buffer. It is up to the programmer to decide when enough of these updates have been batched and to issue the call that turns them into a physical screen update. Picking the right spacing of physical updates can make a great deal of difference to the feel of the graphics client. Both the X server and the *curses*(3) library used by roguelike programs are organized in this way.

Persistent service daemons are a more Unix-specific example of batching. There are two reasons, one obvious and one subtle, to write persistent daemons (as opposed to CLI servers that are started up fresh for each session). The obvious reason is to manage updates to a shared resource. The less obvious reason, which obtains even for daemons that don't handle updates, is to amortize the cost of reading in the daemon's database across multiple requests. A perfect example of this is the DNS service daemon *named*(8), which must sometimes handle thousands of requests per second, each one of which may actually be blocking a user's Web page load. One of the tactics that makes *named*(8) fast is that it replaces parses of expensive on-disk text files describing DNS zones with accesses to a cache held in memory.

12.4.2 Overlapping Operations

In Chapter 5 we compared the POP3 and IMAP protocols for querying remote-mail servers. We noted that IMAP requests (unlike POP3 requests) are tagged with a request identifier generated by the client; the server, when it ships back a response, includes the tag of the request it pertains to.

POP3 requests have to be processed in lockstep by both client and server; the client sends a request, waits for the response to that request, and only then can prepare and ship the next one. IMAP requests, on the other hand, are are tagged so they can be overlapped. If an IMAP client knows that it wants to fetch multiple messages, it can stream several fetch requests (each with a different tag) to the IMAP server, without waiting for responses between them. Responses, each tagged, will come back when the server is ready; responses to early requests may come in while the client is still shipping later ones.

This strategy is general to more areas than network protocols. If you want to cut latency, blocking or waiting on intermediate results is deadly.

12.4.3 Caching Operation Results

Sometimes you can get the best of both worlds (low latency and good throughput) by computing expensive results as needed and caching them for later use. Earlier we mentioned that *named* reduces latency by batching; it also reduces latency by caching the results of previous network transactions with other DNS servers.

Caching has its own problems and tradeoffs, which are well illustrated by one application: the use of binary caches to eliminate parsing overhead associated with text database files. Some variants of Unix have used this technique to speed up access to their password information (the usual motivation was to cut latency on logins at very large sites).

To make this work, all code that looks at the binary cache has to know that it should check the timestamps on both files and regenerate the cache if the text master is newer. Alternatively, all changes to the textual master must be made through a wrapper that will update the binary format.

While this approach can be made to work, it has all the disadvantages that the SPOT rule would lead us to expect. The duplication of data means that it doesn't yield any economy of storage—it's purely a speed optimization. But the real problem with it is that the code to ensure coherency between cache and master is notoriously leaky and bug-prone. Very frequently updated cache files can lead to subtle race conditions simply because of the 1-second resolution of timestamps.

Coherency can be guaranteed in simple cases. One such is the Python interpreter, which compiles and deposits on disk a p-code file with extension `.pyc` when a Python library file is first imported. On subsequent runs the cached copy of the p-code is loaded unless the source has since changed (this avoids re-parsing the library source code on every run). Emacs Lisp uses a similar technique with `.el` and `.elc` files. This technique works because both read and write accesses to the cache go through a single program.

When the update pattern of the master is more complex, however, the synchronization code tends to spring leaks. The Unix variants that used this technique to speed up access to critical system databases were infamous for spawning system-administrator horror stories that reflected this.

In general, binary cache files are a brittle technique and probably best avoided. The work that went into implementing a special-purpose hack to reduce latency in this one case would have been better spent improving the application design so it doesn't have a bottleneck there—or even on tuning to improve the speed of the file system or the virtual-memory implementation.

When you think you are in a situation that demands caching, it is wise to look one level deeper and ask why the caching is necessary. It may well be no more difficult to solve that problem than it would be to get all the edge cases in the caching software right.