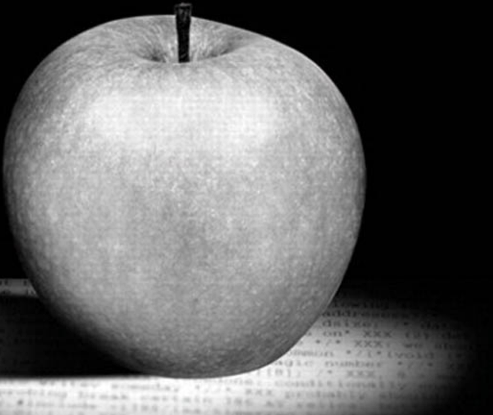
Effective Software Development Series
Scott Meyers, Consulting Editor

Code *Quality*

The Open Source Perspective



Diomidis Spinellis

Code Quality

4

Time Performance

Time is nature's way to keep everything from happening all at once.

— John Archibald Wheeler

A number of software attributes are associated with the behavior of the system across time. The interactions between the various attributes are nontrivial and often involve tradeoffs. Therefore, before embarking on a project to improve the time performance of a program's operation, it is important to determine which of the time-related attributes we want to change. The most important attributes are

Latency Also referred to as *response time*, *wall clock time*, or *execution time*: the time between the start and the completion of an event (for example, between submitting a form and receiving an acknowledgment). Typically, individual computer users want to decrease this measure, as it often affects their productivity by keeping them idle, waiting for an operation to complete.

Throughput Also sometimes referred to as *bandwidth*: the total amount of work done in a given unit of time (for example, transactions or source code lines processed every second). In most cases, system administrators will want to increase this figure, as it measures the utilization of the equipment they manage.

Processor time requirements Also referred to as CPU *time*: a measure of the time the computer's CPU is kept busy, as opposed to waiting for data to arrive from a slower peripheral. This waiting time is important because in many cases, the processor, instead of being idle, can be put into productive use on other jobs, thus increasing the total throughput of an installation.

Real-time response In some cases, the operation of a system may be degraded or be incorrect if the system does not respond to an external event within a (typically short) time interval. Such systems are termed real-time systems: *soft* real-time systems if their operation is degraded by a late response (think of a glitch in an MP3 player); *hard* real-time systems if a late response renders their operation incorrect (think of a cell phone failing to synchronize with its base station). In contrast to the other attributes we describe, real-time response is a Boolean measure: A system may or may not succeed in achieving it.

Time variability Finally, in many systems, we are less interested in specific throughput or latency requirements and more interested in a behavior that does not exhibit time variability. In a video game, for example, we might tolerate different refresh rates for the characters but not a jerky movement.

In addition, when we examine server-class systems, we are interested in how the preceding properties will be affected by changes in the system's workload. In such cases, we also look at performance metrics, such as *load sensitivity*, *capacity*, and *scalability*.

We started this chapter by noting that when setting out to work on the time-related properties of a program, we must have a clear purpose concerning the attributes we might want to improve. Although some operations, such as removing a redundant calculation, may simultaneously improve a number of attributes, other changes often involve tradeoffs. As an example, changing a system to perform transactions in batches may improve the system's throughput and decrease processor time requirements, but, on the other hand, the change will probably introduce higher latency and time variability. Some systems even make such tradeoffs explicit: Sun's JVM runtime invocation options[1] optimize the virtual machine performance for latency or throughput by using different implementations of the garbage collector and the locking primitives. Other programs, such as *tcpdump*[2] and *cat*,[3] provide an option for disabling block buffering on the output stream, allowing the user to obtain lower latency at the expense of decreased throughput. In general, it is easier to improve bandwidth (by throwing more resources at the problem) than latency; historically over every period in which technology doubled the bandwidth of microprocessors, memories, the network, or hard disk, the corresponding latency improvement was no more than by a factor of 1.2 to 1.4. An old network saying captures this as follows:

[1] `-server` and `-client`
[2] netbsdsrc/usr.sbin/tcpdump/tcpdump.c:191–197
[3] netbsdsrc/bin/cat/cat.c:104–106

Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God.

In addition, when examining code with an eye on its performance, it is worthwhile to keep in mind other code attributes that may suffer as a result of any performance-related optimizations.

- ⚠ Many efficient algorithms are a lot more complex than their less efficient counterparts. Any implementation that uses them may see its *reliability* and *readability* suffer. Compare the 135 lines of the C library's optimized quicksort implementation[4] against the relatively inefficient 10-line bubble sort implementation[5] that the X Window System server uses for selecting a display device. Clearly, reimplementing quicksort for selecting between a couple of different screens would have been an overkill. The use of bubble sort is justified in this case, although calling the C library's `qsort` function could have been another, perhaps even better, alternative.
- ⚠ Some optimizations take advantage of a particular platform's characteristics, such as operating system–specific calls or specialized CPU instructions. Such optimizations will negatively impact the code's *portability*. For example, the assembly language implementation of the X Window System's VGA server raster operations[6] is suitable only for a specific CPU architecture and a specific compiler.
- ⚠ Other optimizations rely on developing proprietary communication protocols or file storage formats, thus reducing the system's *interoperability*. As an example, a binary file format[7] may be more efficient to process but a lot less portable than a corresponding XML format.[8]
- ⚠ Finally, performance optimizations often rely on exploiting special cases of a routine's input. The implementation of special cases destroys the code's *simplicity*, *clarity*, and *generality*. To convince yourself, count the number of assumptions made in the special-case handling of single-character key symbols in the internally used Xt library function `StringToKeySym`.[9]

[4]netbsdsrc/lib/libc/stdlib/qsort.c:48–182
[5]XFree86-3.3/xc/programs/Xserver/hw/xfree86/common/xf86Config.c:1160–1159
[6]XFree86-3.3/xc/programs/Xserver/hw/xfree86/vga256/enhanced/vgaFasm.h:77–286
[7]netbsdsrc/games/adventure/save.c
[8]argouml/org/argouml/xml/argo/ArgoParser.java
[9]XFree86-3.3/xc/lib/Xt/TMparse.c:837–842

For all those reasons, the first piece of advice all optimization experts agree on is: Don't optimize; you can see a number of more colorful renditions of the same principle in Figure 4.1. The second piece of advice invariably is: Measure before optimizing. Only by locating a program's bottlenecks will you be able to minimize the human programming cost and the reduction in source code quality typically associated with optimization efforts.

If you are lucky enough to work on a new software system rather than on the code of an existing one, you can apply many best practices from the field of *software performance engineering* throughout your product's lifecycle to ensure that you end up with a responsive and scalable system. This is a large field with a considerable body of knowledge; it would not be possible to cover it in this chapter, but a summary of its most important elements in this paragraph can give you a taste of what it entails (see the Further Reading section at the end of the chapter for more details). On the project management front, you should try to estimate your project's performance risk and set precise quantitative objectives for the project's critical use cases (for example, "a static web page shall be delivered in no more than 50 μs"). When modeling, you will benefit from assessing various design alternatives for your system's architecture so as to avoid expensive mistakes before you commit yourself to code. To do this, you need to build a performance model that will provide you with best- and worst-case estimates of your resource requirements (for example, "the response time increases linearly with the number of transactions). Your guide through this process will be measurement experiments that provide representative and reproducible results and software instrumentation that facilitates the collection of data. Here is an excerpt of measurement code, in its simplest form:[10]

```
void
sendfile(int fd, char *name, char *mode)
{
    startclock();
    [...]
    stopclock();
    printstats("Sent", amount);
}

static void
printstats(const char *direction, unsigned long amount)
{
    [...]
```

[10]netbsdsrc/usr.bin/tftp/tftp.c:95–195, 433–448

The First Rule of Program Optimization: Don't do it.

— Michael A. Jackson

The Second Rule of Program Optimization—for experts only: Don't do it yet.

— Michael A. Jackson

Premature optimization is the root of all evil (or at least most of it) in programming.

— Donald E. Knuth

A fast program is just as important as a correct one—false!

— Steve McConnell

Optimizations always bust things, because all optimizations are, in the long haul, a form of cheating, and cheaters eventually get caught.

— Larry Wall

The key to performance is elegance, not battalions of special cases. The terrible temptation to tweak should be resisted unless the payoff is really noticeable.

— Jon L. Bentley and M. Douglas McIlroy

More computing sins are committed in the name of efficiency than for any other single reason—including blind stupidity.

— William A. Wulf

Improved efficiency comes at the cost of practically every other desirable product attribute.

— Martin Carroll and Margaret Ellis

When the "efficiency" programmers had trouble, they were loath to change their approach because they would then have to sacrifice some efficiency.

— Gerald M. Weinberg

Do not strive to write fast programs—strive to write good ones.

— Joshua Bloch

Figure 4.1 Experts caution against optimizing code

```
    printf("%s %ld bytes in %.1f seconds",
           direction, amount, delta);
    printf(" [%.0f bits/sec]", (amount*8.)/delta);
}
```

On the implementation front, when you consider alternatives, you should always use measurement data to evaluate them; then, when you write the corresponding code, measure its performance-critical components early and often, to avoid nasty surprises later on.

You will read more about measurement techniques in the next section. In many cases, a major source of performance improvements is the use of a more efficient algorithm; this topic is covered in Section 4.2. Having ruled out important algorithmic inefficiencies, you can begin to look for expensive operations that impact your program's performance. Such operations can range from expensive CPU instructions to operating system and peripheral interactions; all are covered in Sections 4.3–4.6. Finally, in Section 4.7, we examine how caching is often used to trade memory space for execution time.

Exercise 4.1 Choose five personal productivity applications and five infrastructure applications your organization relies on. For each application, list its most important time-related attribute.

Exercise 4.2 Your code provides a horrendously complicated yet remarkably efficient implementation of a simple algorithm. Although it works correctly, measurements have demonstrated that a simple call to the language's runtime library implementation would work just as well for all possible input cases. Provide five arguments for scrapping the existing code.

4.1 Measurement Techniques

Humans are notoriously bad at guessing why a system is exhibiting a particular time-related behavior. Starting your search by plunging into the system's source code, looking for the time-wasting culprit, will most likely be a waste of your time. The only reliable and objective way to diagnose and fix time inefficiencies and problems is to use appropriate measurement tools. Even tools, however, can lie when applied to the wrong problem. The best approach, therefore, is to first evaluate and understand the type of workload a program imposes on your system and then use the appropriate tools for each workload type to analyze the problem in detail.