25

Code-Tuning Strategies

CC2E.COM/2578

Contents

Related Topics

Code-tuning techniques: Chapter 29

Software architecture: Section 3.5

THIS CHAPTER DISCUSSES THE QUESTION of performance tuning—historically, a controversial issue. Computer resources were severely limited in the 1960s, and efficiency was a paramount concern. As computers became more powerful in the 1970s, programmers realized how much their focus on performance had hurt readability and maintainability, and code tuning received less attention. The return of performance limitations with the microcomputer revolution of the 1980s again brought efficiency to the fore, which then waned throughout the 1990s. In the 2000s, memory limitations in embedded software for devices such as telephones and PDAs, and the execution time of interpreted code have once again made efficiency a key topic.

You can address performance concerns at two levels: strategic and tactical. This chapter addresses strategic performance issues: what performance is, how important it is, and the general approach to achieving it. If you already have a good grip on performance strategies and are looking for specific code-level techniques that improve performance, move on to the next chapter. Before you begin any major performance work, however, at least skim the information in this chapter so that you don't waste time optimizing when you should be doing other kinds of work.

25.1 Performance Overview

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.
—W.A. Wulf

Code tuning is one way of improving a program's performance. You can often find other ways to improve performance more, in less time and with less harm to the code, than by code tuning. This section describes the options.

Quality Characteristics and Performance

Some people look at the world through rose-colored glasses. Programmers like you and me tend to look at the world through code-colored glasses. We assume that the better we make the code, the more our clients and customers will like our software.

This point of view might have a mailing address somewhere in reality, but it doesn't have a street number, and it certainly doesn't own any real estate. Users are more interested in tangible program characteristics than they are in code quality. Sometimes users are interested in raw performance, but only when it affects their work. Users tend to be more interested in program throughput than raw performance. Delivering software on time, providing a clean user interface, and avoiding downtime are often more significant.

Here's an illustration: I take at least 50 pictures a week on my digital camera. To upload the pictures to my computer, the software that came with the camera requires me to select each picture one by one, viewing them in a window that shows only 6 pictures at a time. Uploading 50 pictures is a tedious process that required dozens of mouse clicks and lots of navigation through the 6-picture window. After putting up with this for a few months, I bought a memory-card reader that plugs directly into my computer and that my computer thinks is a disk drive. Now I can use Windows Explorer to copy the pictures to my computer. What used to take dozens of mouse clicks and lots of waiting now requires about two mouse clicks, a CTRL+A, and a drag and drop.

I really don't care whether the memory card reader transfers each file in half the time or twice the time as the other software, because my throughput is faster. Regardless of whether the memory card reader's code is faster or slower, it's performance is better.

KEY POINT

Performance is only loosely related to code speed. To the extent that you work on your code's speed, you're not working on other quality characteristics. Be wary of sacrificing other characteristics in order to make your code faster. Your work on speed may hurt performance rather than help it.

Performance and Code Tuning

Once you've chosen efficiency as a priority, whether its emphasis is on speed or on size, you should consider several options before choosing to improve either speed or size at the code level. Think about efficiency from each of these viewpoints:

- Program requirements
- System design
- Class and routine design
- Operating-system interactions
- Code compilation
- Hardware
- Code tuning

Program Requirements

Performance is stated as a requirement far more often than it actually is a requirement. Barry Boehm tells the story of a system at TRW that initially required sub-second response time. This requirement led to a highly complex design and an estimated cost of $100 million. Further analysis determined that users would be satisfied with four-second responses 90 percent of the time. Modifying the response-time requirement reduced overall system cost by about $70 million. (Boehm 2000b).

Before you invest time solving a performance problem, make sure that you're solving a problem that needs to be solved.

Program Design

CROSS-REFERENCE For details on designing performance into a program, see the "Additional Resources" section at the end of the chapter.

This level includes the major strokes of the design for a single program, mainly the way in which a program is divided into classes. Some program designs make it difficult to write a high-performance system. Others make it hard not to.

Consider the example of a real-world data-acquisition program for which the high-level design had identified measurement throughput as a key product attribute. Each measurement included time to make an electrical measurement, calibrate the value, scale the value, and convert it from sensor data units (such as millivolts) into engineering data units (such as degrees).

In this case, without addressing the risk in the high-level design, the programmers would have found themselves trying to optimize the math to evaluate a 13th-order polynomial in software—that is, a polynomial with 14 terms including variables raised to the 13th power. Instead, they addressed the problem with different hardware and a high-level design that used dozens of 3rd-order polynomials. This change could not have been effected through code tuning, and it's unlikely that any amount of code tuning would have solved the problem. This is an example of a problem that had to be addressed at the program-design level.

CROSS-REFERENCE For details on the way programmers work toward objectives, see "Setting Objectives" in Section 20.2.

If you know that a program's size and speed are important, design the program's architecture so that you can reasonably meet your size and speed goals. Design a performance-oriented architecture, and then set resource goals for individual subsystems, features, and classes. This will help in several ways:

- Setting individual resource goals makes the system's ultimate performance predictable. If each feature meets its resource goals, the whole system will meet its goals. You can identify subsystems that have trouble meeting their goals early and target them for redesign or code tuning.
- The mere act of making goals explicit improves the likelihood that they'll be achieved. Programmers work to objectives when they know what they are; the more explicit the objectives, the easier they are to work to.

KEY POINT

- You can set goals that don't achieve efficiency directly but promote efficiency in the long run. Efficiency is often best treated in the context of other issues. For example, achieving a high degree of modifiability can provide a better basis for meeting efficiency goals than explicitly setting an efficiency target. With a highly modular, modifiable design, you can easily swap less-efficient components for more-efficient ones.

Class and Routine Design

CROSS-REFERENCE For more information about data types and algorithms, see the "Additional Resources" section at the end of the chapter.

Designing the internals of classes and routines presents another opportunity to design for performance. One key to performance that comes into play at this level is the choice of data types and algorithms, which usually affect both the memory use and the execution speed of a program.

Operating-System Interactions

CROSS-REFERENCE For code-level strategies for dealing with slow or fat operating-system routines, see Chapter 26, "Code-Tuning Techniques."

If your program works with external files, dynamic memory, or output devices, it's probably interacting with the operating system. If performance isn't good, it might be because the operating-system routines are slow or fat. You might not be aware that the program is interacting with the operating system; sometimes your compiler generates system calls or your libraries invoke system calls you would never dream of. More on this later.

Code Compilation

CROSS-REFERENCE The optimization results reported in Chapter 26, “Code-Tuning Techniques,” provide numerous examples of compiler optimizations that produce more efficient code than manual code tuning does.

Good compilers turn clear, high-level language code into optimized machine code. If you choose the right compiler, you might not need to think about optimizing speed any further.

Hardware

Sometimes the cheapest and best way to improve a program’s performance is to buy new hardware. If you’re distributing a program for nationwide use by hundreds of thousands of customers, buying new hardware isn’t a realistic option. But if you’re developing custom software for a few in-house users, a hardware upgrade might be the cheapest option. It saves the cost of initial performance work. It saves the cost of future maintenance problems caused by performance work. It improves the performance of every other program that runs on that hardware too.

Code Tuning

Code tuning is the practice of modifying correct code in ways that make it run more efficiently, and it is the subject of the rest of this chapter. “Tuning” refers to small-scale changes that affect a single class, a single routine, or, more commonly, a few lines of code. “Tuning” does not refer to large-scale design changes, or other higher-level means of improving performance.

You can make dramatic improvements at each level from system design through code tuning. Jon Bentley cites an argument that in some systems, the improvements at each level can be multiplied (1982). Since you can achieve a 10-fold improvement in each of six levels, that implies a potential performance improvement of a million fold. Although such a multiplication of improvements requires a program in which gains at one level are independent of gains at other levels, which is rare, the potential is inspiring.

25.2 Introduction to Code Tuning

What is the appeal of code tuning? It’s not the most effective way to improve performance. Program architecture, class design, and algorithm selection usually produce more dramatic improvements. Nor is it the easiest way to improve performance. Buying new hardware or a compiler with a better optimizer is easier. It’s not the cheapest way to improve performance either. It takes more time to hand-tune code initially, and hand-tuned code is harder to maintain later.

Code tuning is appealing for several reasons. One attraction is that it seems to defy the laws of nature. It’s incredibly satisfying to take a routine that executes in 20 microseconds, tweak a few lines, and reduce the execution speed to 2 microseconds.

It’s also appealing because mastering the art of writing efficient code is a rite of passage to becoming a serious programmer. In tennis, you don’t get any points for the way you pick up a tennis ball, but you still need to learn the right way to do it. You can’t just lean over and pick it up with your hand. If you’re good, you whack it with the head of your racket until it bounces waist high and then you catch it. Whacking it more than three times or not bouncing it the first time are both serious failings. It doesn’t really matter how you pick up a tennis ball, but within the tennis culture the way you pick it up carries a certain cachet. Similarly, no one but you and other programmers usually cares how tight your code is. Nonetheless, within the programming culture, writing micro-efficient code proves you’re cool.

The problem with code tuning is that efficient code isn’t necessarily “better” code. That’s the subject of the next few subsections.

The Pareto Principle

The Pareto Principle, also known as the 80/20 rule, states that you can get 80 percent of the result with 20 percent of the effort. The principle applies to a lot of areas other than programming, but it definitely applies to program optimization.

KEY POINT

Barry Boehm reports that 20 percent of a program’s routines consume 80 percent of its execution time (1987b). In his classic paper “An Empirical Study of Fortran Programs,” Donald Knuth found that less than 4 percent of a program usually accounts for more than 50 percent of its run time (1971).

Knuth used a line-count profiler to discover this surprising relationship, and the implications for optimization are clear. You should measure the code to find the hot spots and then put your resources into optimizing the few percent that are used the most. Knuth profiled his line-count program and found that it was spending half its execution time in two loops. He changed a few lines of code and doubled the speed of the profiler in less than an hour.

Jon Bentley describes a case in which a thousand-line program spent 80 percent of its time in a five-line square-root routine. By tripling the speed of the square-root routine, he doubled the speed of the program (1988).

Bentley also reports the case of a team who discovered that half an operating system’s time was spent in a small loop. They rewrote the loop in microcode and made the loop 10 times faster, but it didn’t change the system’s performance—they had rewritten the system’s idle loop!

The team who designed the ALGOL language—the granddaddy of most modern languages and one of the most influential languages ever—received the following advice: "The best is the enemy of the good." Working toward perfection may prevent completion. Complete it first, and then perfect it. The part that needs to be perfect is usually small.

Old Wives' Tales

Much of what you've heard about code tuning is false. Here are some common misapprehensions:

Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code—false!

Many programmers cling tenaciously to the belief that if they can write code in one or two lines, it will be the most efficient possible. Consider the following code that initializes a 10-element array:

CROSS-REFERENCE Both these code fragments violate several rules of good programming. Readability and maintenance are usually more important than execution speed or size, but in this chapter the topic is performance, and that implies a tradeoff with the other objectives. You'll see many examples of coding practices here that aren't recommended in other parts of this book.

```
for i = 1 to 10
   a[ i ] = i
end for
```

Would you guess that these lines are faster or slower than the following 10 lines that do the same job?

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

If you follow the old "fewer lines are faster" dogma, you'll guess that the first code is faster because it has four fewer lines. Hah! Tests in Visual Basic and Java have shown that the second fragment is at least 60 percent faster than the first. Here are the numbers:

Language	*for*-Loop Time	Straight-Code Time	Time Savings	Performance Ratio
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

Note: (1) Times in this and the following tables in this chapter are given in seconds and are meaningful only for comparisons across rows in each table. Actual times will vary according to the compiler and compiler options used and the environment in which each test is run. (2) Benchmark results are typically made up of several *thousand to many million executions of the code fragments to smooth out sample-to-sample fluctuations in the results. (3) Specific brands and versions of compilers aren't indicated. Performance characteristics vary significantly from brand to brand and from version to version. (4) Comparisons among results from different languages aren't always meaningful because compilers for different languages don't always offer comparable code-generation options. (5) The results shown for interpreted languages (PHP and Python) are typically based on less than 1% of the test runs used for the other languages. (6) Some of the "time savings" percentages might not be exactly reproducible from the data in these tables due to rounding of the "straight time" and "code-tuned time" entries.*

This certainly doesn't imply the conclusion that increasing the number of lines of high-level language code always improves speed or reduces size. It does imply that regardless of the aesthetic appeal of writing something with the fewest lines of code, there's no predictable relationship between the number of lines of code in a high-level language and a program's ultimate size and speed.

Certain operations are probably faster or smaller than others—false! There's no room for "probably" when you're talking about performance. You must always measure performance to know whether your changes helped or hurt your program. The rules of the game change every time you change languages, compilers, versions of compilers, libraries, versions of libraries, processor, amount of memory on the machine, color of shirt you're wearing and so. (These are all serious except the last one.) What was true on one machine with one set of tools can easily be false on another machine with a different set of tools.

This phenomenon suggests several reasons not to improve performance by code tuning. If you want your program to be portable, techniques that improve performance in one environment can degrade it in others. If you change compilers or upgrade, the new compiler might automatically optimize code the way you were hand-tuning it, and your work will have been wasted. Even worse, your code tuning might defeat more powerful compiler optimizations that have been designed to work with straightforward code.

When you tune code, you're implicitly signing up to reprofile each optimization every time you change your compiler brand, compiler version, library version, and so. If you don't reprofile, an optimization that improves performance under one version of a compiler or library might well degrade performance when you change the build environment.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
—Donald Knuth

You should optimize as you go—false!

One theory is that if you strive to write the fastest and smallest possible code as you write each routine, your program will be fast and small. This approach creates a forest-for-the-trees situation in which programmers ignore significant global optimizations because they're too busy with micro-optimizations. Here are the main problems with optimizing as you go along:

- It's almost impossible to identify performance bottlenecks before a program is working completely. Programmers are very bad at guessing which 4 percent of the code accounts for 50 percent of the execution time, and so programmers who optimize as they go will, on average, spend 96 percent of their time optimizing code that doesn't need to be optimized. That leaves very little time to optimize the 4 percent that really counts.
- In the rare case in which developers identify the bottlenecks correctly, they overkill the bottlenecks they've identified and allow others to become critical. Again, the ultimate effect is a reduction in performance. Optimizations done after a system is complete can identify each problem area and its relative importance so that optimization time is allocated effectively.
- Focusing on optimization during initial development detracts from achieving other program objectives. Developers immerse themselves in algorithm analysis and arcane debates that in the end don't contribute much value to the user. Concerns such as correctness, information hiding, and readability become secondary goals, even though performance is easier to improve later than these other concerns are. Post-hoc performance work typically affects less than 5 percent of a program's code. Would you rather go back and do performance work on 5 percent of the code or readability work on 100 percent?

In short, premature optimization's primary drawback is its lack of perspective. Its victims include final code speed, performance attributes that are more important than code speed, program quality, and ultimately the software's users.

If the development time saved by implementing the simplest program is devoted to optimizing the running program, the result will always be a faster-running program than one in which optimization efforts have been exerted indiscriminately as the program was developed (Stevens 1981).

In an occasional project, post-hoc optimization won't be sufficient to meet performance goals, and you'll have to make major changes in the completed code. In those cases, small, localized optimizations wouldn't have provided the gains needed anyway. The problem in such cases isn't inadequate code quality—it's inadequate software architecture.

If you need to optimize before a program is complete, minimize the risks by building perspective into your process. One way is to specify size and speed goals for features and then optimize to meet the goals as you go along. Setting such goals in a specification is a way to keep one eye on the forest while you figure out how big your particular tree is.

FURTHER READING For many other entertaining and enlightening anecdotes, see Gerald Weinberg's *Psychology of Computer Programming* (1998).

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

It's hardly ever true that programs need to be fast or small before they need to be correct. Gerald Weinberg tells the story of a programmer who was flown to Detroit to help debug a troubled program. The programmer worked with the team who had developed the program and concluded after several days that the situation was hopeless.

On the flight home, he mulled over the situation and realized what the problem was. By the end of the flight, he had an outline for the new code. He tested the code for several days and was about to return to Detroit when he got a telegram saying that the project had been cancelled because the program was impossible to write. He headed back to Detroit anyway and convinced the executives that the project could be completed.

Then he had to convince the project's original programmers. They listened to his presentation, and when he'd finished, the creator of the old system asked, "And how long does your program take?"

"That varies, but about ten seconds per input."

"Aha! But my program takes only one second per input." The veteran leaned back, satisfied that he'd stumped the upstart. The other programmers seemed to agree, but the new programmer wasn't intimidated.

"Yes, but your program *doesn't work*. If mine doesn't have to work, I can make it run instantly."

For a certain class of projects, speed or size is a major concern. This class is the minority, is much smaller than most people think, and is getting smaller all the time. For these projects, the performance risks must be addressed by up-front design. For other projects, early optimization poses a significant threat to overall software quality, *including performance*.

When to Tune

Jackson's Rules of Optimization: Rule 1. Don't do it. Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution.
—M. A. Jackson

Use a high-quality design. Make the program right. Make it modular and easily modifiable so that it's easy to work on later. When it's complete and correct, check the performance. If the program lumbers, make it fast and small. Don't optimize until you know you need to.

A few years ago I worked on a C++ project that produced graphical outputs to analyze investment data. After my team got the first graph working, testing reported that the program took about 45 minutes to draw the graph, which was clearly not acceptable. We held a team meeting to decide what to do about it. One of the developers became irate and shouted, "If we want to have any chance of releasing an acceptable product we've got to start rewriting the whole code base in assembler *right now*." I responded that I didn't think so—that 4 percent of the code probably accounted for 50 percent or more of the performance bottleneck. It would be best to address that 4 percent toward the end of the project. After a bit more shouting, our manager assigned me to do some initial performance work (which was really a case of "Oh no! Please don't throw me into that briar patch!").

As is often the case, a day's work identified a couple of glaring bottlenecks in the code, and a small number of code-tuning changes reduced the drawing time from 45 minutes to less than 30 seconds. Far less than 1 percent of the code accounted for 90 percent of the run time. By the time we released the software months later, several additional code-tuning changes reduced that drawing time to a little more than 1 second.

Compiler Optimizations

Modern compiler optimizations might be more powerful than you expect. In the case I described earlier, my compiler did as good a job of optimizing a nested loop as I was able to do by rewriting the code in a supposedly more efficient style.

When shopping for a compiler, compare the performance of each compiler on your program. Each compiler has different strengths and weaknesses, and some will be better suited to your program than others.

Optimizing compilers are better at optimizing straightforward code than they are at optimizing tricky code. If you do "clever" things like fooling around with loop indexes, your compiler has a harder time doing its job and your program suffers. See "Using Only One Statement per Line" in Section 31.5, for an example in which a straightforward approach resulted in compiler-optimized code that was 11 percent faster than comparable "tricky" code.

With a good optimizing compiler, your code speed can improve 40 percent or more across the board. Many of the techniques described in the next chapter produce gains of only 15-30 percent. Why not just write clear code and let the compiler do the work? Here are the results of a few tests to check how much an optimizer speeded up an insertion-sort routine:

Language	Time Without Compiler Optimizations	Time with Compiler Optimizations	Time Savings	Performance Ratio
C++ compiler 1	2.21	1.05	52%	2:1
C++ compiler 2	2.78	1.15	59%	2.5:1
C++ compiler 3	2.43	1.25	49%	2:1
C# compiler	1.55	1.55	0%	1:1
Visual Basic	1.78	1.78	0%	1:1
Java VM 1	2.77	2.77	0%	1:1
Java VM 2	1.39	1.38	<1%	1:1
Java VM 3	2.63	2.63	0%	1:1

The only difference between versions of the routine was that compiler optimizations were turned off for the first compile, on for the second. Clearly, some compilers optimize better than others, and some are better without optimizations in the first place. Some JVMs are also clearly better than others. You'll have to check your own compiler, JVM, or both to measure its effect.

25.3 Kinds of Fat and Molasses

In code tuning you find the parts of a program that are as slow as molasses in winter and as big as Godzilla and change them so that they run like greased lightning and are so skinny they can hide in the cracks between the other bytes in RAM. You always have to profile the program to know with any confidence which parts are slow and fat, but some operations have a long history of laziness and obesity, and you can start by investigating them.

Common Sources of Inefficiency

Here are several common sources of inefficiency:

Input/output operations

One of the most significant sources of inefficiency is unnecessary I/O. If you have a choice of working with a file in memory vs. on disk, in a database, or across a network, use an in-memory data unless space is critical.

Here's a performance comparison between code that accesses random elements in a 100-element in-memory array and code that accesses random elements of the same size in a 100-record disk file:

Language	External File Time	In-Memory Data Time	Time Savings	Performance Ratio
C++	6.04	0.000	100%	n/a
C#	12.8	0.010	100%	1000:1

According to this data, in-memory access is on the order of 1000 times faster than accessing data in an external file. Indeed with the C++ compiler I used, the time required for in-memory access wasn't measurable.

The performance comparison for a similar test of sequential access times is similar:

Language	External File Time	In-Memory Data Time	Time Savings	Performance Ratio
C++	3.29	0.021	99%	150:1
C#	2.60	0.030	99%	85:1

The tests for sequential access were run with 13 times the data volume of the tests for random access, so the results are not comparable across the two types of tests.

If the test had used a slower medium for external access—hard disk across a network connection—the difference would have been even greater. Here is what the performance looks like when a similar random-access test is performed on a network location instead of on the local machine:

Language	Local File Time	Network File Time	Time Savings
C++	6.04	6.64	-10%
C#	12.8	14.1	-10%

Of course these results can vary dramatically depending on the speed of your network, network loading, distance of the local machine from the networked disk drive, speed of the networked disk drive compared to the speed of the local drive, current phase of the moon, and other factors.

Overall, the effect of in-memory access is significant enough to make you think twice about having I/O in a speed-critical part of a program.

Paging

An operation that causes the operating system to swap pages of memory is much slower than an operation that works on only one page of memory. Sometimes a simple change makes a huge difference. In the next example, one programmer wrote an initialization loop that produced many page faults on a system that used 4K pages.

Java Example of an Initialization Loop That Causes Many Page Faults

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {
   for ( row = 0; row < MAX_ROWS; row++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

This is a nicely formatted loop with good variable names, so what's the problem? The problem is that each element of *table* is about 4000 bytes long. If *table* has too many rows, every time the program accesses a different row, the operating system will have to switch memory pages. The way the loop is structured, every single array access switches rows, which means that every single array access causes paging to disk.

The programmer restructured the loop this way:

Java Example of an Initialization Loop That Causes Few Page Faults

```
for ( row = 0; row < MAX_ROWS; row++ ) {
   for ( column = 0; column < MAX_COLUMNS; column++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

This code still causes a page fault every time it switches rows, but it switches rows only *MAX_ROWS* times instead of *MAX_ROWS* * *MAX_COLUMNS* times.

The specific performance penalty varies significantly. On a machine with limited memory, I measured the second code sample to be about 1000 times faster than the first code sample. On machines with more memory, I've measured the difference to be as small as a factor of 2, and it doesn't show up at all except for very large values of *MAX_ROWS* and *MAX_COLUMNS*.

System calls

Calls to system routines are often expensive. System routines include input/output operations to disk, keyboard, screen, printer, or other device; memory-management routines; and certain utility routines. If performance is an issue, find out how expensive your system calls are. If they're expensive, consider these options:

- Write your own services. Sometimes you need only a small part of the functionality offered by a system routine and can build your own from lower-level system routines. Writing your own replacement gives you something that's faster, smaller, and better suited to your needs.

- Avoid going to the system.
- Work with the system vendor to make the call faster. Most vendors want to improve their products and are glad to learn about parts of their systems with weak performance. (They may seem a little grouchy about it at first, but they really are interested.)

In the code tuning initiative I describe in the "When to Tune" Section, the program used an *AppTime* class that was derived from a commercially available *BaseTime* class. (These names have been changed to protect the guilty.) The *AppTime* object was the most common object in this application, and we instantiated tens of thousands of *AppTime* objects. After several months, we discovered that *BaseTime* was initializing itself to the system time in its constructor. For our purposes, the system time was irrelevant, which meant we were needlessly generating thousands of system-level calls. Simply overriding *BaseTime*'s constructor and initializing the *time* field to 0 instead of to the system time gave us about as much performance improvement as all the other changes we made put together.

Interpreted languages

Interpreted languages tend to exact significant performance penalties because they must process each programming-language instruction before creating and executing machine code. In the performance benchmarking I performed for this chapter and Chapter 26, I observed the following approximate relationships in performance among different languages:

Table 25-1. Relative execution time of programming languages

Language	Type of Language	Execution time relative to C++
C++	Compiled	1:1
Visual Basic	Compiled	1:1
C#	Compiled	1:1
Java	Byte code	1.5:1
PHP	Interpreted	>100:1
Python	Interpreted	>100:1

As you can see from the table, C++, Visual Basic, and C# are all comparable. Java is close, but tends to be slower than the other languages. PHP and Python are interpreted languages, and code in those languages tended to run a factor of 100 or more slower than code in C++, VB, C#, and Java.

The general numbers presented in this table must be viewed cautiously. For any particular piece of code, C++, VB, C#, or Java might be twice as fast or half as fast as the other languages. (You can see this for yourself in the detailed examples in Chapter 26.)

Errors

A final source of performance problems is errors in the code. Errors can include leaving debugging code turned on (such as logging trace information to a file), forgetting to deallocate memory, improperly designing database tables, and so on.

A version 1.0 application I worked on had a particular operation that was much slower than other similar operations. A great deal of project mythology grew up to explain the slowness of this operation. We released version 1.0 without ever fully understanding why this particular operation was so slow. While working on the version 1.1 release, however, I discovered that the database table used by the operation wasn't indexed! Simply indexing the table improved performance by a factor of 30 for some operations. Defining an index on a commonly-used table is not optimization; it's just good programming practice.

Relative Performance Costs of Common Operations

Although you can't count on some operations being more expensive than others without measuring them, certain operations tend to be more expensive. When you look for the molasses in your program, use Table 25-2 to help make some initial guesses about the sticky parts of your program.

Table 25-2. Costs of Common Operations

		Relative Time Consumed	
Operation	**Example**	**C++**	**Java**
Baseline (integer assignment)	*i = j*	*1*	*1*
Routine Calls			
Call routine with no parameters	*foo()*	1	n/a
Call private routine with no parameters	*this.foo()*	1	0.5
Call private routine with 1 parameter	*this.foo(i)*	1.5	0.5
Call private routine with 2 parameters	*this.foo(i, j)*	1.7	0.5
Object routine call	*bar.foo()*	2	1
Derived routine call	*derivedBar.foo()*	2	1
Polymorphic routine call	*abstractBar.foo()*	2.5	2
Object References			
Level 1 object dereference	*i = obj.num*	1	1

		Relative Time Consumed	
Operation	**Example**	**C++**	**Java**
Level 2 object dereference	*i = obj1.obj2. num*	1	1
Each additional dereference	*i = obj1.obj2.obj3...*	not measurable	not measurable
Integer Operations			
Integer assignment (local)	*i = j*	1	1
Integer assignment (inherited)	*i = j*	1	1
Integer addition	*i = j + k*	1	1
Integer subtraction	*i = j + k*	1	1
Integer multiplication	*i = j * k*	1	1
Integer division	*i = j % k*	5	1.5
Floating Point Operations			
Floating-point assignment	*x = y*	1	1
Floating-point addition	*x = y + z*	1	1
Floating-point subtraction	*x = y - z*	1	1
Floating-point multiplication	*x = y * z*	1	1
Floating-point division	*x = y / z*	4	1
Transcendental Functions			
Floating-point square root	*x = sqrt(y)*	15	4
Floating-point sine	*x = sin(y)*	25	20
Floating-point logarithm	*x = log(y)*	25	20
Floating-point e^x	*x = exp(y)*	50	20
Arrays			
Access integer array with constant subscript	*i = a[5]*	1	1
Access integer array with variable subscript	*i = a[j]*	1	1
Access two-dimensional integer array with constant subscripts	*i = a[3, 5]*	1	1
Access two-dimensional integer array with variable subscripts	*i = a[j, k]*	1	1
Access floating-point array with constant subscript	*x = z[5]*	1	1

		Relative Time Consumed	
Operation	**Example**	**C++**	**Java**
Access floating-point array with integer-variable subscript	*x = z[j]*	1	1
Access two-dimensional floating-point array with constant subscripts	*x = z[3, 5]*	1	1
Access two-dimensional floating-point array with integer-variable subscripts	*x = z[j, k]*	1	1

Note: Measurements in this table are highly sensitive to local machine environment, compiler optimizations, and code generated by specific compilers. Measurements between C++ and Java are not directly comparable.

Relative performance of these operations has changed significantly since the first edition of *Code Complete*, so if you're approaching code tuning with 10-year-old ideas about performance, you might need to update your thinking.

Most of the common operations are about the same price—routine calls, assignments, integer arithmetic, and floating-point arithmetic are all roughly equal. Transcendental math functions are extremely expensive. Polymorphic routine calls are a bit more expensive than other kinds of routine calls.

This table, or a similar one that you make, is the key that unlocks all the speed improvements described in Chapter 26, "Code-Tuning Techniques." In every case, improving speed comes from replacing an expensive operation with a cheaper one. The next chapter provides examples of how to do so.

25.4 Measurement

Since small parts of a program usually consume a disproportionate share of the run time, measure your code to find the hot spots.

Once you've found the hot spots and optimized them, measure the code again to assess how much you've improved it. Many aspects of performance are counterintuitive. The earlier case in this chapter, in which 10 lines of code were significantly faster and smaller than one line, is one example of the ways that code can surprise you.

KEY POINT

Experience doesn't help much with optimization either. A person's experience might have come from an old machine, language, or compiler—and when any of those things changes, all bets are off. You can never be sure about the effect of an optimization until you measure the effect.

A few years ago I wrote a program that summed the elements in a matrix. The original code looked like the next example.

C++ Example of Straightforward Code to Sum the Elements in a Matrix

```
sum = 0;
for ( row = 0; row < rowCount; row++ ) {
   for ( column = 0; column < columnCount; column++ ) {
      sum = sum + matrix[ row ][ column ];
   }
}
```

This code was straightforward, but performance of the matrix-summation routine was critical, and I knew that all the array accesses and loop tests had to be expensive. I knew from computer-science classes that every time the code accessed a two-dimensional array, it performed expensive multiplications and additions. For a 100-by-100 matrix, that totaled 10,000 multiplications and additions plus the loop overhead. By converting to pointer notation, I reasoned, I could increment a pointer and replace 10,000 expensive multiplications with 10,000 relatively cheap increment operations. I carefully converted the code to pointer notation and got this:

C++ Example of an Attempt to Tune Code to sum the Elements in a Matrix

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;
while ( elementPointer < lastElementPointer ) {
   sum = sum + *elementPointer++;
}
```

FURTHER READING Jon Bentley reported a similar experience in which converting to pointers hurt performance by about 10 percent. The same conversion had—in another setting—improved performance more than 50 percent. See "Software Exploratorium: Writing Efficient C Programs" (Bentley 1991).

Even though the code wasn't as readable as the first code, especially to programmers who aren't C++ experts, I was magnificently pleased with myself. For a 100-by-100 matrix, I calculated that I had saved 10,000 multiplications and a lot of loop overhead. I was so pleased that I decided to measure the speed improvement, something I didn't always do back then, so that I could pat myself on the back more quantitatively.

No programmer has* ever *been able to predict or analyze where performance bottlenecks are* without data. *No matter where you think it's going, you will be surprised to discover that it is going somewhere else.
—Joseph M. Newcomer

Do you know what I found?

No improvement whatsoever. Not with a 100-by-100 matrix. Not with a 10-by-10 matrix. Not with any size matrix. I was so disappointed that I dug into the assembly code generated by the compiler to see why my optimization hadn't worked. To my surprise, it turned out that I was not the first programmer who ever needed to iterate through the elements of an array—the compiler's optimizer was already converting the array accesses to pointers. I learned that the only result of optimization you can usually be sure of without measuring performance is that you've made your code harder to read. If it's not worth measuring to know that it's more efficient, it's not worth sacrificing clarity for a performance gamble.

Measurements Need to be Precise

CROSS-REFERENCE For a discussion of profiling tools, see "Code Tuning" in Section 30.3.

Performance measurements need to be precise. Timing your program with a stopwatch or by counting "one elephant, two elephant, three elephant" isn't precise enough. Profiling tools are useful, or you can use your system's clock and routines that record the elapsed times for computing operations.

Whether you use someone else's tool or write your own code to make the measurements, make sure that you're measuring only the execution time of the code you're tuning. Use the number of CPU clock ticks allocated to your program rather than the time of day. Otherwise, when the system switches from your program to another program, one of your routines will be penalized for the time spent executing another program. Likewise, try to factor out measurement overhead so that neither the original code nor the tuning attempt is unfairly penalized.

25.5 Iteration

Once you've identified a performance bottleneck, you'll be amazed at how much you can improve performance by code tuning. You'll rarely get a 10-fold improvement from one technique, but you can effectively combine techniques; so keep trying, even after you find one that works.

I once wrote a software implementation of the Data Encryption Standard, or DES. Actually, I didn't write it once—I wrote it about 30 times. Encryption according to DES encodes digital data so that it can't be unscrambled without a password. The encryption algorithm is so convoluted that it seems like it's been used on itself. The performance goal for my DES implementation was to encrypt an 18K file in 37 seconds on an original IBM PC. My first implementation executed in 21 minutes and 40 seconds, so I had a long row to hoe.

Even though most individual optimizations were small, cumulatively they were significant. To judge from the percentage improvements, no three or even four optimizations would have met my performance goal. But the final combination was effective. The moral of the story is that if you dig deep enough, you can make some surprising gains.

The code tuning I did in this case is the most aggressive code tuning I've ever done. At the same time, the final code is the most unreadable, unmaintainable code I've ever written. The initial algorithm is complicated. The code resulting from the high-level language transformation was barely readable. The translation to assembler produced a single 500-line routine that I'm afraid to look at. In general, this relationship between code tuning and code quality holds true. Here's a table that shows a history of the optimizations:

CROSS-REFERENCE The techniques listed in this table are described in Chapter 26, "Code-Tuning Techniques."

Optimization	Benchmark Time	Improvement
Implement initially—straightforward	21:40	—
Convert from bit fields to arrays	7:30	65%
Unroll innermost *for* loop	6:00	20%
Remove final permutation	5:24	10%
Combine two variables	5:06	5%
Use a logical identity to combine the first two steps of the DES algorithm	4:30	12%
Make two variables share the same memory to reduce data shuttling in inner loop	3:36	20%
Make two variables share the same memory to reduce data shuttling in outer loop	3:09	13%
Unfold all loops and use literal array subscripts	1:36	49%
Remove routine calls and put all the code in line	0:45	53%
Rewrite the whole routine in assembler	0:22	51%
Final	**0:22**	**98%**

Note: The steady progress of optimizations in this table doesn't imply that all optimizations work. I haven't shown all the things I tried that doubled the run time. At least two-thirds of the optimizations I tried didn't work.

25.6 Summary of the Approach to Code Tuning

Here are the steps you should take as you consider whether code tuning can help you improve the performance of a program:

1. Develop the software using well-designed code that's easy to understand and modify.
2. If performance is poor,
    a. Save a working version of the code so that you can get back to the "last known good state."
    b. Measure the system to find hot spots.
    c. Determine whether the weak performance comes from inadequate design, data types, or algorithms and whether code tuning is appropriate. If code tuning isn't appropriate, go back to step 1.
    d. Tune the bottleneck identified in step (c).
    e. Measure each improvement one at a time.
    f. If an improvement doesn't improve the code, revert to the code saved in step (a). (Typically, more than half the attempted tunings will produce only a negligible improvement in performance or degrade performance.)
3. Repeat from step 2.

CC2E.COM/2585

Additional Resources

Performance

Smith, Connie U. and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Boston, Mass.: Addison Wesley, 2002. This book covers software performance engineering, an approach for building performance into software systems at all stages of development. It makes extensive use of examples and case studies for several kinds of programs. It includes specific recommendations for web applications and pays special attention to scalability.

CC2E.COM/2592

Newcomer, Joseph M. "Optimization: Your Worst Enemy," May 2000, *www.flounder.com/optimization.htm*. Newcomer is an experienced systems programmer who describes the various pitfalls of ineffective optimization strategies in graphic detail.

Algorithms and Data Types

CC2E.COM/2599

Knuth, Donald. *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, 3d ed. Reading, Mass.: Addison-Wesley, 1997.

Knuth, Donald. *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, 3d ed. Reading, Mass.: Addison-Wesley, 1997.

Knuth, Donald. *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, 2d ed. Reading, Mass.: Addison-Wesley, 1998.

These are the first three volumes of a series that was originally intended to grow to seven volumes. They can be somewhat intimidating. In addition to the English description of the algorithms, they're described in mathematical notation or MIX, an assembly language for the imaginary MIX computer. The books contain exhaustive details on a huge number of topics, and if you have an intense interest in a particular algorithm, you won't find a better reference.

Sedgewick, Robert. *Algorithms in Java, Parts 1-4, 3d ed.* Boston, Mass.: Addison-Wesley, 2002. This book's four parts contain a survey of the best methods of solving a wide variety of problems. Its subject areas include fundamentals, sorting, searching, abstract data type implementation, and advanced topics. Sedgewick's *Algorithms in Java, Part 5, 3d ed.* (2003) covers graph algorithms. Sedgewick's *Algorithms in C++, Parts 1-4, 3d ed.* (1998), *Algorithms in C++, Part 5, 3d ed.* (2002), *Algorithms in C, Parts 1-4, 3d ed.* (1997), *and Algorithms in C, Part 5, 3d ed.* (2001) are similarly organized. Sedgewick was a Ph.D. student of Knuth's.

CC2E.COM/2506

CHECKLIST: Code-Tuning Strategy

Overall Program Performance

- ❑ Have you considered improving performance by changing the program requirements?
- ❑ Have you considered improving performance by modifying the program's design?
- ❑ Have you considered improving performance by modifying the class design?
- ❑ Have you considered improving performance by avoiding operating system interactions?

- ❑ Have you considered improving performance by avoiding I/O?
- ❑ Have you considered improving performance by using a compiled language instead of an interpreted language?
- ❑ Have you considered improving performance by using compiler optimizations?
- ❑ Have you considered improving performance by switching to different hardware?
- ❑ Have you considered code tuning only as a last resort?

Code-Tuning Approach

- ❑ Is your program fully correct before you begin code tuning?
- ❑ Have you measured performance bottlenecks before beginning code tuning?
- ❑ Have you measured the effect of each code-tuning change
- ❑ Have you backed out the code-tuning changes that didn't produce the intended improvement?
- ❑ Have you tried more than one change to improve performance of each bottleneck, i.e., *iterated*?

Key Points

- Performance is only one aspect of overall software quality, and it's usually not the most important. Finely tuned code is only one aspect of overall performance, and it's usually not the most significant. Program architecture, detailed design, and data-structure and algorithm selection usually have more influence on a program's execution speed and size than the efficiency of its code does.
- Quantitative measurement is a key to maximizing performance. It's needed to find the areas in which performance improvements will really count, and it's needed again to verify that optimizations improve rather than degrade the software.
- Most programs spend most of their time in a small fraction of their code. You won't know which code that is until you measure it.
- Multiple iterations are usually needed to achieve desired performance improvements through code tuning.
- The best way to prepare for performance work during initial coding is to write clean code that's easy to understand and modify.