CHAPTER 5

Optimizing Program Performance

The biggest speedup you'll ever get with a program will be when you first get it working.

—John K. Ousterhout

The primary objective in writing a program must be to make it work correctly under all possible conditions. A program that runs fast but gives incorrect results serves no useful purpose. Programmers must write clear and concise code, not only so that they can make sense of it, but also so that others can read and understand the code during code reviews and when modifications are required later.

On the other hand, there are many occasions when making a program run fast is also an important consideration. If a program must process video frames or network packets in real time, then a slow-running program will not provide the needed functionality. When a computation task is so demanding that it requires days or weeks to execute, then making it run just 20% faster can have significant impact. In this chapter, we will explore how to make programs run faster via several different types of program optimization.

Writing an efficient program requires several types of activities. First, we must select an appropriate set of algorithms and data structures. Second, we must write source code that the compiler can effectively optimize to turn into efficient executable code. For this second part, it is important to understand the capabilities and limitations of optimizing compilers. Seemingly minor changes in how a program is written can make large differences in how well a compiler can optimize it. Some programming languages are more easily optimized than others. Some features of C, such as the ability to perform pointer arithmetic and casting, make it challenging for a compiler to optimize. Programmers can often write their programs in ways that make it easier for compilers to generate efficient code. A third technique for dealing with especially demanding computations is to divide a task into portions that can be computed in parallel, on some combination of multiple cores and multiple processors. We will defer this aspect of performance enhancement to Chapter 12. Even when exploiting parallelism, it is important that each parallel thread execute with maximum performance, and so the material of this chapter remains relevant in any case.

In approaching program development and optimization, we must consider how the code will be used and what critical factors affect it. In general, programmers must make a trade-off between how easy a program is to implement and maintain, and how fast it runs. At an algorithmic level, a simple insertion sort can be programmed in a matter of minutes, whereas a highly efficient sort routine may take a day or more to implement and optimize. At the coding level, many low-level optimizations tend to reduce code readability and modularity, making the programs more susceptible to bugs and more difficult to modify or extend. For code that will be executed repeatedly in a performance-critical environment, extensive optimization may be appropriate. One challenge is to maintain some degree of elegance and readability in the code despite extensive transformations.

We describe a number of techniques for improving code performance. Ideally, a compiler would be able to take whatever code we write and generate the most

efficient possible machine-level program having the specified behavior. Modern compilers employ sophisticated forms of analysis and optimization, and they keep getting better. Even the best compilers, however, can be thwarted by *optimization blockers*—aspects of the program's behavior that depend strongly on the execution environment. Programmers must assist the compiler by writing code that can be optimized readily.

The first step in optimizing a program is to eliminate unnecessary work, making the code perform its intended task as efficiently as possible. This includes eliminating unnecessary function calls, conditional tests, and memory references. These optimizations do not depend on any specific properties of the target machine.

To maximize the performance of a program, both the programmer and the compiler require a model of the target machine, specifying how instructions are processed and the timing characteristics of the different operations. For example, the compiler must know timing information to be able to decide whether it should use a multiply instruction or some combination of shifts and adds. Modern computers use sophisticated techniques to process a machine-level program, executing many instructions in parallel and possibly in a different order than they appear in the program. Programmers must understand how these processors work to be able to tune their programs for maximum speed. We present a high-level model of such a machine based on recent designs of Intel and AMD processors. We also devise a graphical *data-flow* notation to visualize the execution of instructions by the processor, with which we can predict program performance.

With this understanding of processor operation, we can take a second step in program optimization, exploiting the capability of processors to provide *instruction-level parallelism*, executing multiple instructions simultaneously. We cover several program transformations that reduce the data dependencies between different parts of a computation, increasing the degree of parallelism with which they can be executed.

We conclude the chapter by discussing issues related to optimizing large programs. We describe the use of code *profilers*—tools that measure the performance of different parts of a program. This analysis can help find inefficiencies in the code and identify the parts of the program on which we should focus our optimization efforts. Finally, we present an important observation, known as *Amdahl's law*, which quantifies the overall effect of optimizing some portion of a system.

In this presentation, we make code optimization look like a simple linear process of applying a series of transformations to the code in a particular order. In fact, the task is not nearly so straightforward. A fair amount of trial-and-error experimentation is required. This is especially true as we approach the later optimization stages, where seemingly small changes can cause major changes in performance, while some very promising techniques prove ineffective. As we will see in the examples that follow, it can be difficult to explain exactly why a particular code sequence has a particular execution time. Performance can depend on many detailed features of the processor design for which we have relatively little documentation or understanding. This is another reason to try a number of different variations and combinations of techniques.