A program has two basic resources at its disposal:

- Processor time (i.e., CPU cycles)
- Memory

In this chapter, I will demonstrate ways to make efficient use of processor time—which is to say that I’ll show you ways to make your program do the most, in the smallest amount of time, by pruning away unnecessary operations. Don’t forget Vilfredo Pareto’s heuristic, which I introduced in the last chapter. Unless your code is a complete mess, most of your performance
problems will arise in a small number of places. Always use a profiler to identify bottlenecks so that you don't make the mistake of blindly optimizing everything. Optimization makes code brittle. If you optimize your entire program, you are likely to drastically undermine its structural integrity. As I mentioned in the last chapter, it's like making a pact with Lucifer. Sure, your code will be faster, but only at a steep price.

**WARNING** In this section, more than a few of the techniques that I present will directly contradict some of the things I talked about in the previous chapter. This is because optimization involves trade-offs. No perfect solution exists for every problem, but rather solutions that are successful under certain conditions. You can make your program smaller, but it will probably cost you CPU cycles. Likewise, you can make your program faster if you don't mind putting up with memory bloat. Rarely can you have your cake and eat it too. Instead, you will need to strike a balance between speed and size. The approach that you adopt ultimately will depend upon your priorities.

### 6.1 Program Control Jumps

The fastest way for a processor to execute operations is sequentially, one after another. Naturally, when you start forcing the processor to jump around from one place to the next in memory, everything goes to hell. Not only can a jump instruction require the processor to leave the confines of the cache, but it usually also entails the overhead of setting up an activation record.

#### 6.1.1 Labels and GOTO

Consider the following function definition:

```c
int orderEntry(
    char *customerID,
    char *productID,
    int quantity,
    float total,
    char *salesRep
)
{
    //perform order entry
}
```

The sheer overhead involved in calling this function is nontrivial. You can see this by looking at the Intel assembly code used to call this function: