# 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.

Studying the assembly-code representation of a program is one of the most effective means for gaining an understanding of the compiler and how the generated code will run. A good strategy is to start by look-

ing carefully at the code for the inner loops, identifying performance-reducing attributes such as excessive memory references and poor use of registers. Starting with the assembly code, we can also predict what operations will be performed in parallel and how well they will use the processor resources. As we will see, we can often determine the time (or at least a lower bound on the time) required to execute a loop by identifying *critical paths*, chains of data dependencies that form during repeated executions of a loop. We can then go back and modify the source code to try to steer the compiler toward more efficient implementations.

Most major compilers, including GCC, are continually being updated and improved, especially in terms of their optimization abilities. One useful strategy is to do only as much rewriting of a program as is required to get it to the point where the compiler can then generate efficient code. By this means, we avoid compromising the readability, modularity, and portability of the code as much as if we had to work with a compiler of only minimal capabilities. Again, it helps to iteratively modify the code and analyze its performance both through measurements and by examining the generated assembly code.

To novice programmers, it might seem strange to keep modifying the source code in an attempt to coax the compiler into generating efficient code, but this is indeed how many high-performance programs are written. Compared to the alternative of writing code in assembly language, this indirect approach has the advantage that the resulting code will still run on other machines, although perhaps not with peak performance.

## 5.1 Capabilities and Limitations of Optimizing Compilers

Modern compilers employ sophisticated algorithms to determine what values are computed in a program and how they are used. They can then exploit opportunities to simplify expressions, to use a single computation in several different places, and to reduce the number of times a given computation must be performed. Most compilers, including GCC, provide users with some control over which optimizations they apply. As discussed in Chapter 3, the simplest control is to specify the *optimization level*. For example, invoking GCC with the command-line flag '`-O1`' will cause it to apply a basic set of optimizations. As discussed in Web Aside ASM:OPT, invoking GCC with flag '`-O2`' or '`-O3`' will cause it to apply more extensive optimizations. These can further improve program performance, but they may expand the program size and they may make the program more difficult to debug using standard debugging tools. For our presentation, we will mostly consider code compiled with optimization level 1, even though optimization level 2 has become the accepted standard for most GCC users. We purposely limit the level of optimization to demonstrate how different ways of writing a function in C can affect the efficiency of the code generated by a compiler. We will find that we can write C code that, when compiled just with optimization level 1, vastly outperforms a more naïve version compiled with the highest possible optimization levels.

Compilers must be careful to apply only *safe* optimizations to a program, meaning that the resulting program will have the exact same behavior as would an unoptimized version for all possible cases the program may encounter, up to the limits of the guarantees provided by the C language standards. Constraining the compiler to perform only safe optimizations eliminates possible sources of undesired run-time behavior, but it also means that the programmer must make more of an effort to write programs in a way that the compiler can then transform into efficient machine-level code. To appreciate the challenges of deciding which program transformations are safe or not, consider the following two procedures:

```
1 void twiddle1(int *xp, int *yp)
```

```
2 {
3     *xp += *yp;
4     *xp += *yp;
5 }
6
7 void twiddle2(int *xp, int *yp)
8 {
9     *xp += 2* *yp;
10 }
```

At first glance, both procedures seem to have identical behavior. They both add twice the value stored at the location designated by pointer `yp` to that designated by pointer `xp`. On the other hand, function `twiddle2` is more efficient. It requires only three memory references (read `*xp`, read `*yp`, write `*xp`), whereas `twiddle1` requires six (two reads of `*xp`, two reads of `*yp`, and two writes of `*xp`). Hence, if a compiler is given procedure `twiddle1` to compile, one might think it could generate more efficient code based on the computations performed by `twiddle2`.

Consider, however, the case in which `xp` and `yp` are equal. Then function `twiddle1` will perform the following computations:

```
3     *xp += *xp;  /* Double value at xp */
4     *xp += *xp;  /* Double value at xp */
```

The result will be that the value at `xp` will be increased by a factor of 4. On the other hand, function `twiddle2` will perform the following computation:

```
9     *xp += 2* *xp;  /* Triple value at xp */
```

The result will be that the value at `xp` will be increased by a factor of 3. The compiler knows nothing about how `twiddle1` will be called, and so it must assume that arguments `xp` and `yp` can be equal. It therefore cannot generate code in the style of `twiddle2` as an optimized version of `twiddle1`.

The case where two pointers may designate the same memory location is known as *memory aliasing*. In performing only safe optimizations, the compiler must assume that different pointers may be aliased. As another example, for a program with pointer variables `p` and `q`, consider the following code sequence:

```
x = 1000; y = 3000;
*q = y;   /* 3000 */
*p = x;   /* 1000 */
t1 = *q;  /* 1000 or 3000 */
```

The value computed for `t1` depends on whether or not pointers `p` and `q` are aliased—if not, it will equal 3000, but if so it will equal 1000. This leads to one of the major *optimization blockers*, aspects of programs that can severely limit the opportunities for a compiler to generate optimized code. If a compiler cannot determine whether or not two pointers may be aliased, it must assume that either case is possible, limiting the set of possible optimizations.

**Practice Problem 5.1**:

The following problem illustrates the way memory aliasing can cause unexpected program behavior. Consider the following procedure to swap two values:

```
1 /* Swap value x at xp with value y at yp */
2 void swap(int *xp, int *yp)
3 {
4     *xp = *xp + *yp; /* x+y       */
5     *yp = *xp - *yp; /* x+y-y = x */
6     *xp = *xp - *yp; /* x+y-x = y */
7 }
```

If this procedure is called with `xp` equal to `yp`, what effect will it have?

A second optimization blocker is due to function calls. As an example, consider the following two procedures:

```
1 int f();
2
3 int func1() {
4     return f() + f() + f() + f();
5 }
6
7 int func2() {
8     return 4*f();
9 }
```

It might seem at first that both compute the same result, but with `func2` calling `f` only once, whereas `func1` calls it four times. It is tempting to generate code in the style of `func2` when given `func1` as the source.

Consider, however, the following code for `f`:

```
1 int counter = 0;
2
3 int f() {
4     return counter++;
5 }
```

This function has a *side effect*—it modifies some part of the global program state. Changing the number of times it gets called changes the program behavior. In particular, a call to `func1` would return $0+1+2+3 = 6$, whereas a call to `func2` would return $4 \cdot 0 = 0$, assuming both started with global variable `counter` set to 0.

Most compilers do not try to determine whether a function is free of side effects and hence is a candidate for optimizations such as those attempted in `func2`. Instead, the compiler assumes the worst case and leaves function calls intact.

**Aside: Optimizing function calls by inline substitution**

As described in Web Aside ASM:OPT, code involving function calls can be optimized by a process known as *inline substitution* (or simply "inlining"), where the function call is replaced by the code for the body of the function. For example, we can expand the code for `func1` by substituting four instantiations of function `f`:

```
1 /* Result of inlining f in func1 */
2 int func1in() {
3     int t = counter++;  /* +0 */
4     t += counter++;     /* +1 */
5     t += counter++;     /* +2 */
6     t += counter++;     /* +3 */
7     return t;
8 }
```

This transformation both reduces the overhead of the function calls and allows further optimization of the expanded code. For example, the compiler can consolidate the updates of global variable `counter` in `func1in` to generate an optimized version of the function:

```
1 /* Optimization of inlined code */
2 int func1opt() {
3     int t = 4 * counter + 6;
4     counter = t + 4;
5     return t;
6 }
```

This code faithfully reproduces the behavior of `func1` for this particular definition of function `f`.

Recent versions of GCC attempt this form of optimization, either when directed to with the command-line option '`-finline`' or for optimization levels 2 or higher. Since we are considering optimization level 1 in our presentation, we will assume that the compiler does not perform inline substitution. **End Aside.**

Among compilers, GCC is considered adequate, but not exceptional, in terms of its optimization capabilities. It performs basic optimizations, but it does not perform the radical transformations on programs that more "aggressive" compilers do. As a consequence, programmers using GCC must put more effort into writing programs in a way that simplifies the compiler's task of generating efficient code.

## 5.2 Expressing Program Performance

We introduce the metric *cycles per element*, abbreviated "CPE," as a way to express program performance in a way that can guide us in improving the code. CPE measurements help us understand the loop performance of an iterative program at a detailed level. It is appropriate for programs that perform a repetitive computation, such as processing the pixels in an image or computing the elements in a matrix product.

The sequencing of activities by a processor is controlled by a clock providing a regular signal of some frequency, usually expressed in *gigahertz* (GHz), billions of cycles per second. For example, when product literature characterizes a system as a "4 GHz" processor, it means that the processor clock runs at $4.0 \times 10^9$ cycles per second. The time required for each clock cycle is given by the reciprocal of the clock frequency. These typically are expressed in *nanoseconds* (1 nanosecond is $10^{-9}$ seconds), or *picoseconds* (1 picosecond

```
1  /* Compute prefix sum of vector a */
2  void psum1(float a[], float p[], long int n)
3  {
4      long int i;
5      p[0] = a[0];
6      for (i = 1; i < n; i++)
7          p[i] = p[i-1] + a[i];
8  }
9
10 void psum2(float a[], float p[], long int n)
11 {
12     long int i;
13     p[0] = a[0];
14     for (i = 1; i < n-1; i+=2) {
15         float mid_val = p[i-1] + a[i];
16         p[i]    = mid_val;
17         p[i+1]  = mid_val + a[i+1];
18     }
19     /* For odd n, finish remaining element */
20     if (i < n)
21         p[i] = p[i-1] + a[i];
22 }
```

Figure 5.1: **Prefix-sum functions.** These provide examples for how we express program performance.

is $10^{-12}$ seconds). For example, the period of a 4 GHz clock can be expressed as either 0.25 nanoseconds or 250 picoseconds. From a programmer's perspective, it is more instructive to express measurements in clock cycles rather than nanoseconds or picoseconds. That way, the measurements express how many instructions are being executed rather than how fast the clock runs.

Many procedures contain a loop that iterates over a set of elements. For example, functions `psum1` and `psum2` in Figure 5.1 both compute the *prefix sum* of a vector of length $n$. For a vector $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, the prefix sum $\vec{p} = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ is defined as

$$\begin{array}{rcll} p_0 & = & a_0 & \\ p_i & = & p_{i-1} + a_i, & 1 \leq i < n \end{array} \tag{5.1}$$

Function `psum1` computes one element of the result vector per iteration. The second uses a technique known as *loop unrolling* to compute two elements per iteration. We will explore the benefits of loop unrolling later in this chapter. See Problems 5.11, 5.12, and 5.21 for more about analyzing and optimizing the prefix-sum computation.

The time required by such a procedure can be characterized as a constant plus a factor proportional to the number of elements processed. For example, Figure 5.2 shows a plot of the number of clock cycles required by the two functions for a range of values of $n$. Using a *least squares fit*, we find that the run times (in clock cycles) for `psum1` and `psum2` can be approximated by the equations $496 + 10.0n$ and $500 + 6.5n$, respectively. These equations indicate an overhead of 496 to 500 cycles due to the timing code and to initiate the procedure, set up the loop, and complete the procedure, plus a linear factor of 6.5 or 10.0 cycles

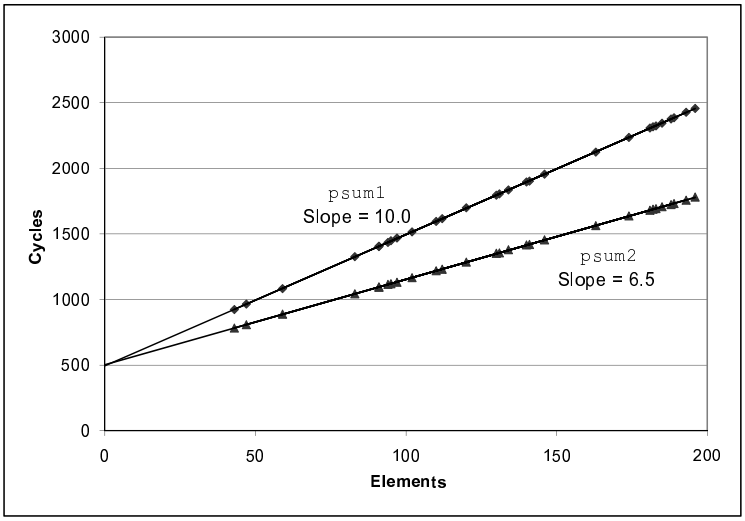



Figure 5.2: **Performance of prefix-sum functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

per element. For large values of $n$ (say, greater than 200), the run times will be dominated by the linear factors. We refer to the coefficients in these terms as the effective number of *cycles per element*, abbreviated "CPE." We prefer measuring the number of cycles per *element* rather than the number of cycles per *iteration*, because techniques such as loop unrolling allow us to use fewer iterations to complete the computation, but our ultimate concern is how fast the procedure will run for a given vector length. We focus our efforts on minimizing the CPE for our computations. By this measure, `psum2`, with a CPE of 6.50, is superior to `psum1`, with a CPE of 10.0.

**Aside: What is a least squares fit?**
For a set of data points $(x_1, y_1), \ldots (x_n, y_n)$, we often try to draw a line that best approximates the X-Y trend represented by this data. With a least squares fit, we look for a line of the form $y = mx + b$ that minimizes the following error measure:

$$E(m, b) \quad = \quad \sum_{i=1,n} (mx_i + b - y_i)^2$$

An algorithm for computing $m$ and $b$ can be derived by finding the derivatives of $E(m, b)$ with respect to $m$ and $b$ and setting them to 0. **End Aside.**

### Practice Problem 5.2:

Later in this chapter we will start with a single function and generate many different variants that preserve the function's behavior, but with different performance characteristics. For three of these variants, we found that the run times (in clock cycles) can be approximated by the following functions:

**Version 1:** $60 + 35n$

**Version 2:** $136 + 4n$

**Version 3:** $157 + 1.25n$

For what values of $n$ would each version be the fastest of the three? Remember that $n$ will always be an integer.

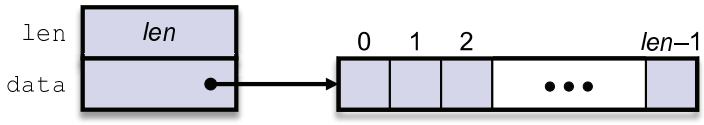



Figure 5.3: **Vector abstract data type.** A vector is represented by header information plus array of designated length.

## 5.3 Program Example

To demonstrate how an abstract program can be systematically transformed into more efficient code, we will use a running example based on the vector data structure shown in Figure 5.3. A vector is represented with two blocks of memory: the header and the data array. The header is a structure declared as follows:

*code/opt/vec.h*

```
1 /* Create abstract data type for vector */
2 typedef struct {
3     long int len;
4     data_t *data;
5 } vec_rec, *vec_ptr;
```

*code/opt/vec.h*

The declaration uses data type `data_t` to designate the data type of the underlying elements. In our evaluation, we measure the performance of our code for integer (C `int`), single-precision floating-point (C `float`), and double-precision floating-point (C `double`) data. We do this by compiling and running the program separately for different type declarations, such as the following for data type `int`:

```
typedef int data_t;
```

We allocate the data array block to store the vector elements as an array of `len` objects of type `data_t`.

Figure 5.4 shows some basic procedures for generating vectors, accessing vector elements, and determining the length of a vector. An important feature to note is that `get_vec_element`, the vector access routine, performs bounds checking for every vector reference. This code is similar to the array representations used in many other languages, including Java. Bounds checking reduces the chances of program error, but it can also slow down program execution.

As an optimization example, consider the code shown in Figure 5.5, which combines all of the elements in a vector into a single value according to some operation. By using different definitions of compile-time constants `IDENT` and `OP`, the code can be recompiled to perform different operations on the data. In particular, using the declarations

```
#define IDENT 0
#define OP  +
```

it sums the elements of the vector. Using the declarations

*code/opt/vec.c*

```
1  /* Create vector of specified length */
2  vec_ptr new_vec(long int len)
3  {
4      /* Allocate header structure */
5      vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
6      if (!result)
7          return NULL;  /* Couldn't allocate storage */
8      result->len = len;
9      /* Allocate array */
10     if (len > 0) {
11         data_t *data = (data_t *)calloc(len, sizeof(data_t));
12         if (!data) {
13             free((void *) result);
14             return NULL; /* Couldn't allocate storage */
15         }
16         result->data = data;
17     }
18     else
19         result->data = NULL;
20     return result;
21 }
22
23 /*
24  * Retrieve vector element and store at dest.
25  * Return 0 (out of bounds) or 1 (successful)
26  */
27 int get_vec_element(vec_ptr v, long int index, data_t *dest)
28 {
29     if (index < 0 || index >= v->len)
30         return 0;
31     *dest = v->data[index];
32     return 1;
33 }
34
35 /* Return length of vector */
36 long int vec_length(vec_ptr v)
37 {
38     return v->len;
39 }
```

*code/opt/vec.c*

Figure 5.4: **Implementation of vector abstract data type.** In the actual program, data type `data_t` is declared to be `int`, `float`, or `double`

.

```
1 /* Implementation with maximum use of data abstraction */
2 void combine1(vec_ptr v, data_t *dest)
3 {
4     long int i;
5
6     *dest = IDENT;
7     for (i = 0; i < vec_length(v); i++) {
8         data_t val;
9         get_vec_element(v, i, &val);
10        *dest = *dest OP val;
11    }
12 }
```

Figure 5.5: **Initial implementation of combining operation.** Using different declarations of identity element *IDENT* and combining operation *OP*, we can measure the routine for different operations.

```
#define IDENT 1
#define OP  *
```

it computes the product of the vector elements.

In our presentation, we will proceed through a series of transformations of the code, writing different versions of the combining function. To gauge progress, we will measure the CPE performance of the functions on a machine with an Intel Core i7 processor, which we will refer to as our *reference machine*. Some characteristics of this processor were given in Section 3.1. These measurements characterize performance in terms of how the programs run on just one particular machine, and so there is no guarantee of comparable performance on other combinations of machine and compiler. However, we have compared the results with those for a number of different compiler/processor combinations, and found them quite comparable.

As we proceed through a set of transformations, we will find that many lead to only minimal performance gains, while others have more dramatic effects. Determining which combinations of transformations to apply is indeed part of the "black art" of writing fast code. Some combinations that do not provide measurable benefits are indeed ineffective, while others are important as ways to enable further optimizations by the compiler. In our experience, the best approach involves a combination of experimentation and analysis: repeatedly attempting different approaches, performing measurements, and examining the assembly-code representations to identify underlying performance bottlenecks.

As a starting point, the following are CPE measurements for `combine1` running on our reference machine, trying all combinations of data type and combining operation. For single-precision and double-precision floating-point data, our experiments on this machine gave identical performance for addition, but differing performance for multiplication. We therefore report five CPE values: integer addition and multiplication, floating-point addition, single-precision multiplication (labeled "F `*`"), and double-precision multiplication (labeled "D `*`").

```
1 /* Move call to vec_length out of loop */
2 void combine2(vec_ptr v, data_t *dest)
3 {
4     long int i;
5     long int length = vec_length(v);
6
7     *dest = IDENT;
8     for (i = 0; i < length; i++) {
9         data_t val;
10        get_vec_element(v, i, &val);
11        *dest = *dest OP val;
12    }
13 }
```

Figure 5.6: **Improving the efficiency of the loop test.** By moving the call to `vec_length` out of the loop test, we eliminate the need to execute it on every iteration.

| Function | Page | Method | Integer | | Floating point | | |
|---|---|---|---|---|---|---|---|
| | | | + | * | + | F * | D * |
| `combine1` | 459 | Abstract unoptimized | 29.02 | 29.21 | 27.40 | 27.90 | 27.36 |
| `combine1` | 459 | Abstract `-O1` | 12.00 | 12.00 | 12.00 | 12.01 | 13.00 |

We can see that our measurements are somewhat imprecise. The more likely CPE number for integer sum and product is 29.00, rather than 29.02 or 29.21. Rather than "fudging" our numbers to make them look good, we will present the measurements we actually obtained. There are many factors that complicate the task of reliably measuring the precise number of clock cycles required by some code sequence. It helps when examining these numbers to mentally round the results up or down by a few hundredths of a clock cycle.

The unoptimized code provides a direct translation of the C code into machine code, often with obvious inefficiencies. By simply giving the command-line option '`-O1`', we enable a basic set of optimizations. As can be seen, this significantly improves the program performance—more than a factor of two—with no effort on behalf of the programmer. In general, it is good to get into the habit of enabling at least this level of optimization. For the remainder of our measurements, we use optimization levels 1 and higher in generating and measuring our programs.

## 5.4 Eliminating Loop Inefficiencies

Observe that procedure `combine1`, as shown in Figure 5.5, calls function `vec_length` as the test condition of the `for` loop. Recall from our discussion of how to translate code containing loops into machine-level programs (Section 3.6.5) that the test condition must be evaluated on every iteration of the loop. On the other hand, the length of the vector does not change as the loop proceeds. We could therefore compute the vector length only once and use this value in our test condition.

Figure 5.6 shows a modified version called `combine2`, which calls `vec_length` at the beginning and

assigns the result to a local variable `length`. This transformation has noticeable effect on the overall performance for some data types and operations, and minimal or even none for others. In any case, this transformation is required to eliminate inefficiencies that would become bottlenecks as we attempt further optimizations.

| Function | Page | Method | Integer | | Floating point | | |
|---|---|---|---|---|---|---|---|
| | | | + | * | + | F * | D * |
| `combine1` | 459 | Abstract `-O1` | 12.00 | 12.00 | 12.00 | 12.01 | 13.00 |
| `combine2` | 460 | Move `vec_length` | 8.03 | 8.09 | 10.09 | 11.09 | 12.08 |

This optimization is an instance of a general class of optimizations known as *code motion*. They involve identifying a computation that is performed multiple times, (e.g., within a loop), but such that the result of the computation will not change. We can therefore move the computation to an earlier section of the code that does not get evaluated as often. In this case, we moved the call to `vec_length` from within the loop to just before the loop.

Optimizing compilers attempt to perform code motion. Unfortunately, as discussed previously, they are typically very cautious about making transformations that change where or how many times a procedure is called. They cannot reliably detect whether or not a function will have side effects, and so they assume that it might. For example, if `vec_length` had some side effect, then `combine1` and `combine2` could have different behaviors. To improve the code, the programmer must often help the compiler by explicitly performing code motion.

As an extreme example of the loop inefficiency seen in `combine1`, consider the procedure `lower1` shown in Figure 5.7. This procedure is styled after routines submitted by several students as part of a network programming project. Its purpose is to convert all of the uppercase letters in a string to lower case. The procedure steps through the string, converting each uppercase character to lowercase. The case conversion involves shifting characters in the range '`A`' to '`Z`' to the range '`a`' to '`z`.'

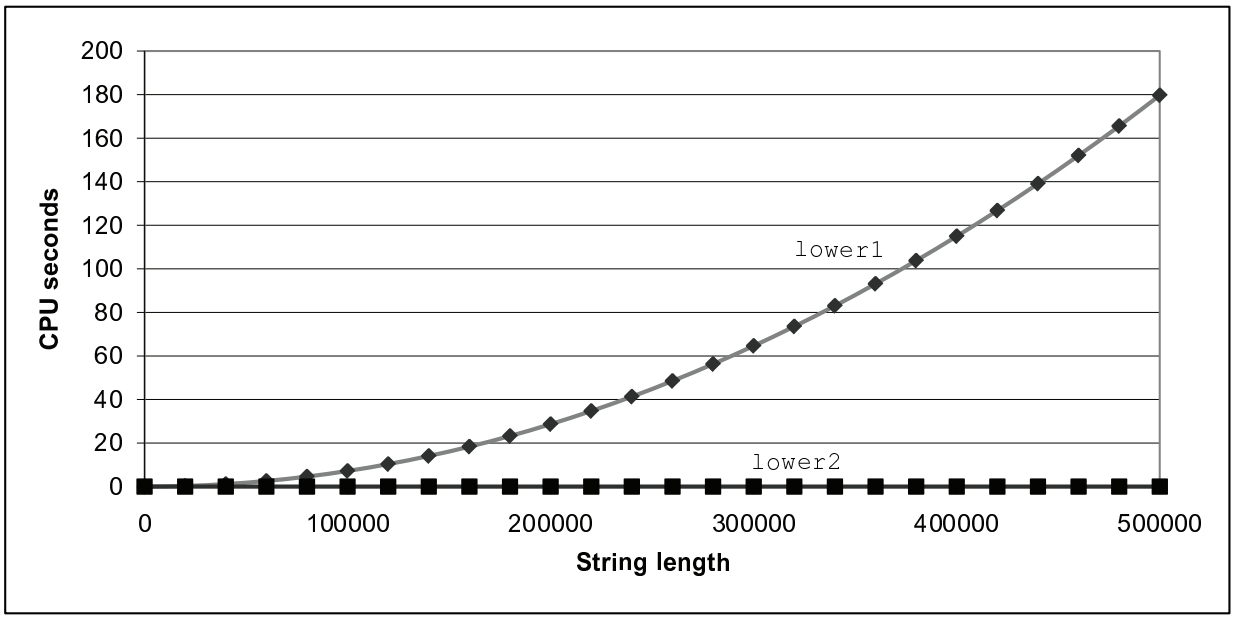
The library function `strlen` is called as part of the loop test of `lower1`. Although `strlen` is typically implemented with special x86 string-processing instructions, its overall execution is similar to the simple version that is also shown in Figure 5.7. Since strings in C are null-terminated character sequences, `strlen` can only determine the length of a string by stepping through the sequence until it hits a null character. For a string of length $n$, `strlen` takes time proportional to $n$. Since `strlen` is called in each of the $n$ iterations of `lower1`, the overall run time of `lower1` is quadratic in the string length, proportional to $n^2$.

This analysis is confirmed by actual measurements of the functions for different length strings, as shown in Figure 5.8 (and using the library version of `strlen`). The graph of the run time for `lower1` rises steeply as the string length increases (Figure 5.8(a)). Figure Figure 5.8(b) shows the run times for seven different lengths (not the same as shown in the graph), each of which is a power of 2. Observe that for `lower1` each doubling of the string length causes a quadrupling of the run time. This is a clear indicator of a quadratic run time. For a string of length 1,048,576, `lower1` requires over 13 minutes of CPU time.

Function `lower2` shown in Figure 5.7 is identical to that of `lower1`, except that we have moved the call to `strlen` out of the loop. The performance improves dramatically. For a string length of 1,048,576, the function requires just 1.5 milliseconds—over 500,000 times faster than `lower1`. Each doubling of the string length causes a doubling of the run time—a clear indicator of linear run time. For longer strings, the run-time improvement will be even greater.

```
1  /* Convert string to lowercase: slow */
2  void lower1(char *s)
3  {
4      int i;
5
6      for (i = 0; i < strlen(s); i++)
7          if (s[i] >= 'A' && s[i] <= 'Z')
8              s[i] -= ('A' - 'a');
9  }
10
11 /* Convert string to lowercase: faster */
12 void lower2(char *s)
13 {
14     int i;
15     int len = strlen(s);
16
17     for (i = 0; i < len; i++)
18         if (s[i] >= 'A' && s[i] <= 'Z')
19             s[i] -= ('A' - 'a');
20 }
21
22 /* Sample implementation of library function strlen */
23 /* Compute length of string */
24 size_t strlen(const char *s)
25 {
26     int length = 0;
27     while (*s != '\0') {
28         s++;
29         length++;
30     }
31     return length;
32 }
```

Figure 5.7: **Lowercase conversion routines.** The two procedures have radically different performance.




(a)

| Function | String length | | | | | | |
|---|---|---|---|---|---|---|---|
| | 16,384 | 32,768 | 65,536 | 131,072 | 262,144 | 524,288 | 1,048,576 |
| `lower1` | 0.19 | 0.77 | 3.08 | 12.34 | 49.39 | 198.42 | 791.22 |
| `lower2` | 0.0000 | 0.0000 | 0.0001 | 0.0002 | 0.0004 | 0.0008 | 0.0015 |

(b)

Figure 5.8: **Comparative performance of lowercase conversion routines.** The original code `lower1` has a quadratic run time due to an inefficient loop structure. The modified code `lower2` has a linear run time.