Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`combine1`	543	Abstract unoptimized	22.68	20.02	19.98	20.18
`combine1`	543	Abstract `-O1`	10.12	10.12	10.17	11.14

We can see that our measurements are somewhat imprecise. The more likely CPE number for integer sum is 23.00, rather than 22.68, while the number for integer product is likely 20.0 instead of 20.02. 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 2—with no effort on behalf of the programmer. In general, it is good to get into the habit of enabling some level of optimization. (Similar performance results were obtained with optimization level `-Og`.) For the remainder of our measurements, we use optimization levels `-O1` and `-O2` when 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 3.6.7) 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`. It 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 +	Integer *	Floating point +	Floating point *
`combine1`	543	Abstract `-O1`	10.12	10.12	10.17	11.14
`combine2`	545	Move `vec_length`	7.02	9.03	9.02	11.03

This optimization is an instance of a general class of optimizations known as *code motion*. They involve identifying a computation that is performed multiple

```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long 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.

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.

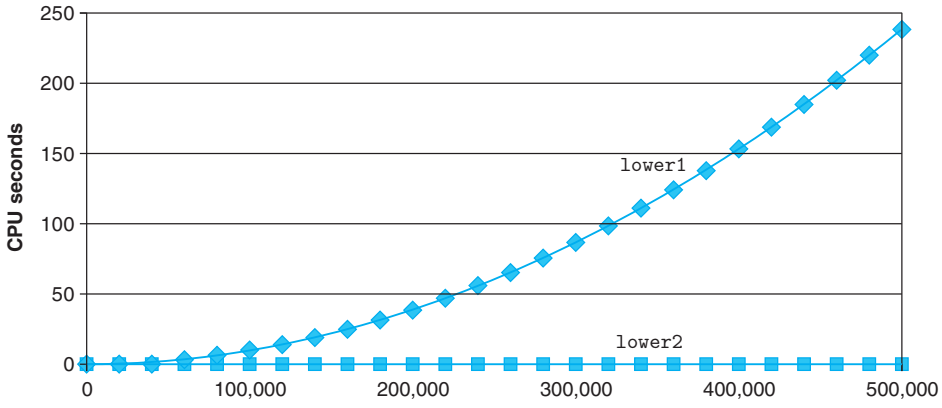
As an extreme example of the loop inefficiency seen in `combine1`, consider the procedure `lower1` shown in Figure 5.7. This procedure is styled after routines submitted by several students as part of a network programming project. Its purpose is to convert all of the uppercase letters in a string to lowercase. 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.

```
1   /* Convert string to lowercase: slow */
2   void lower1(char *s)
3   {
4       long 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      long i;
15      long 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      long 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.

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 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 17 minutes of CPU time.



(a)

	String length						
Function	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
`lower1`	0.26	1.03	4.10	16.41	65.62	262.48	1,049.89
`lower2`	0.0000	0.0001	0.0001	0.0003	0.0005	0.0010	0.0020

(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.

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 2.0 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.

In an ideal world, a compiler would recognize that each call to `strlen` in the loop test will return the same result, and thus the call could be moved out of the loop. This would require a very sophisticated analysis, since `strlen` checks the elements of the string and these values are changing as `lower1` proceeds. The compiler would need to detect that even though the characters within the string are changing, none are being set from nonzero to zero, or vice versa. Such an analysis is well beyond the ability of even the most sophisticated compilers, even if they employ inlining, and so programmers must do such transformations themselves.

This example illustrates a common problem in writing programs, in which a seemingly trivial piece of code has a hidden asymptotic inefficiency. One would not expect a lowercase conversion routine to be a limiting factor in a program's performance. Typically, programs are tested and analyzed on small data sets, for which the performance of `lower1` is adequate. When the program is ultimately

deployed, however, it is entirely possible that the procedure could be applied to strings of over one million characters. All of a sudden this benign piece of code has become a major performance bottleneck. By contrast, the performance of `lower2` will be adequate for strings of arbitrary length. Stories abound of major programming projects in which problems of this sort occur. Part of the job of a competent programmer is to avoid ever introducing such asymptotic inefficiency.

Practice Problem 5.3 **(solution page 609)**

Consider the following functions:

```
long min(long x, long y) { return x < y ? x : y; }
long max(long x, long y) { return x < y ? y : x; }
void incr(long *xp, long v) { *xp += v; }
long square(long x) { return x*x; }
```

The following three code fragments call these functions:

A.
```
for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);
```

B.
```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

C.
```
long low = min(x, y);
long high = max(x, y);

for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

Assume `x` equals 10 and `y` equals 100. Fill in the following table indicating the number of times each of the four functions is called in code fragments A–C:

Code	`min`	`max`	`incr`	`square`
A.				
B.				
C.				

5.5 Reducing Procedure Calls

As we have seen, procedure calls can incur overhead and also block most forms of program optimization. We can see in the code for `combine2` (Figure 5.6) that `get_vec_element` is called on every loop iteration to retrieve the next vector element. This function checks the vector index `i` against the loop bounds with every vector reference, a clear source of inefficiency. Bounds checking might be a useful feature when dealing with arbitrary array accesses, but a simple analysis of the code for `combine2` shows that all references will be valid.