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 these data. With a least squares fit, we look for a line of the form $y = mx + b$ that minimizes the following error measure:

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

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.

procedure, plus a linear factor of 6.0 or 9.0 cycles 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. 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 6.0, is superior to `psum1`, with a CPE of 9.0.

Practice Problem 5.2 (solution page 609)

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.

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:

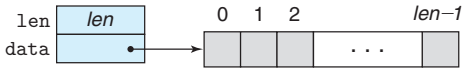

Figure 5.3 Vector abstract data type. A vector is represented by header information plus an array of designated length.

code/opt/vec.h

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

code/opt/vec.h

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

```
typedef long 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

```
#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,

code/opt/vec.c

```
1  /* Create vector of specified length */
2  vec_ptr new_vec(long len)
3  {
4      /* Allocate header structure */
5      vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
6      data_t *data = NULL;
7      if (!result)
8          return NULL;  /* Couldn't allocate storage */
9      result->len = len;
10     /* Allocate array */
11     if (len > 0) {
12         data = (data_t *)calloc(len, sizeof(data_t));
13         if (!data) {
14             free((void *) result);
15             return NULL; /* Couldn't allocate storage */
16         }
17     }
18     /* Data will either be NULL or allocated array */
19     result->data = data;
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 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 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`, `long`, `float`, or `double`.

```
1   /* Implementation with maximum use of data abstraction */
2   void combine1(vec_ptr v, data_t *dest)
3   {
4       long 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.

we measured the CPE performance of the functions on a machine with an Intel Core i7 Haswell processor, which we 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 we have found them generally consistent with those presented here.

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 table shows CPE measurements for `combine1` running on our reference machine, with different combinations of operation (addition or multiplication) and data type (long integer and double-precision floating point). Our experiments with many different programs showed that operations on 32-bit and 64-bit integers have identical performance, with the exception of code involving division operations. Similarly, we found identical performance for programs operating on single- or double-precision floating-point data. In our tables, we will therefore show only separate results for integer data and for floating-point data.

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