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, 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 4.0×10^9 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 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

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

Function `psum1` computes one element of the result vector per iteration. Function `psum2` 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 5.19 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 $368 + 9.0n$ and $368 + 6.0n$, respectively. These equations indicate an overhead of 368 cycles due to the timing code and to initiate the procedure, set up the loop, and complete the

```
1   /* Compute prefix sum of vector a */
2   void psum1(float a[], float p[], long n)
3   {
4       long 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 n)
11  {
12      long 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 even n, finish remaining element */
20      if (i < n)
21          p[i] = p[i-1] + a[i];
22  }
```

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

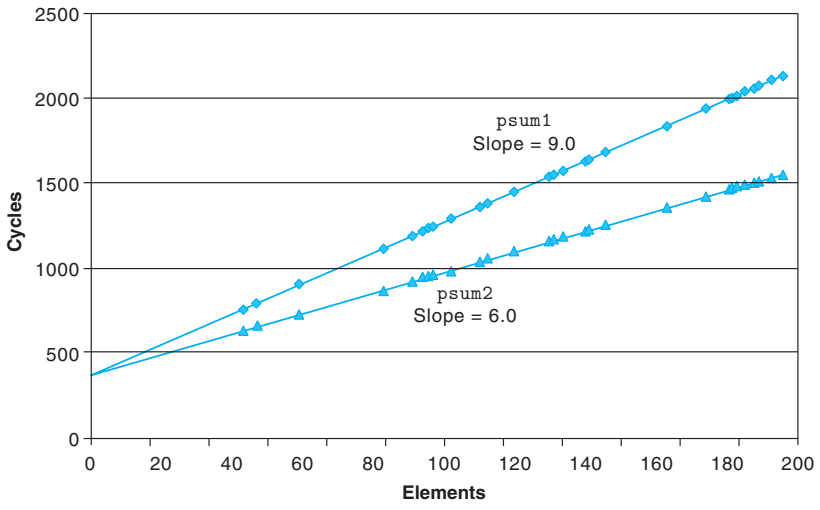

Figure 5.2 **Performance of prefix-sum functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

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: