CHAPTER 5

# Optimizing Program Performance

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 computational 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 execu-

tion 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.

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 and 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 looking 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 option `-Og` specifies that it should apply a basic set of optimizations.

Invoking GCC with option `-O1` or higher (e.g., `-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 `-O1`, even though level `-O2` has become the accepted standard for most software projects that use GCC. 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 option `-O1`, vastly outperforms a more naive 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(long *xp, long *yp)
2   {
3       *xp += *yp;
4       *xp += *yp;
5   }
6
7   void twiddle2(long *xp, long *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 3,000, but if so it will equal 1,000. 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 (solution page 609)

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(long *xp, long *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  long f();
2
3  long func1() {
4      return f() + f() + f() + f();
5  }
6
7  long 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  long counter = 0;
2
3  long 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

**Aside** Optimizing function calls by inline substitution

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  long func1in() {
3      long 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  long func1opt() {
3      long t = 4 * counter + 6;
4      counter += 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 level `-O1` and higher. Unfortunately, GCC only attempts inlining for functions defined within a single file. That means it will not be applied in the common case where a set of library functions is defined in one file but invoked by functions in other files.

There are times when it is best to prevent a compiler from performing inline substitution. One is when the code will be evaluated using a symbolic debugger, such as GDB, as described in Section 3.10.2. If a function call has been optimized away via inline substitution, then any attempt to trace or set a breakpoint for that call will fail. The second is when evaluating the performance of a program by profiling, as is discussed in Section 5.14.1. Calls to functions that have been eliminated by inline substitution will not be profiled correctly.

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 zero.

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.

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 $4.0 \times 10^9$ 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 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.

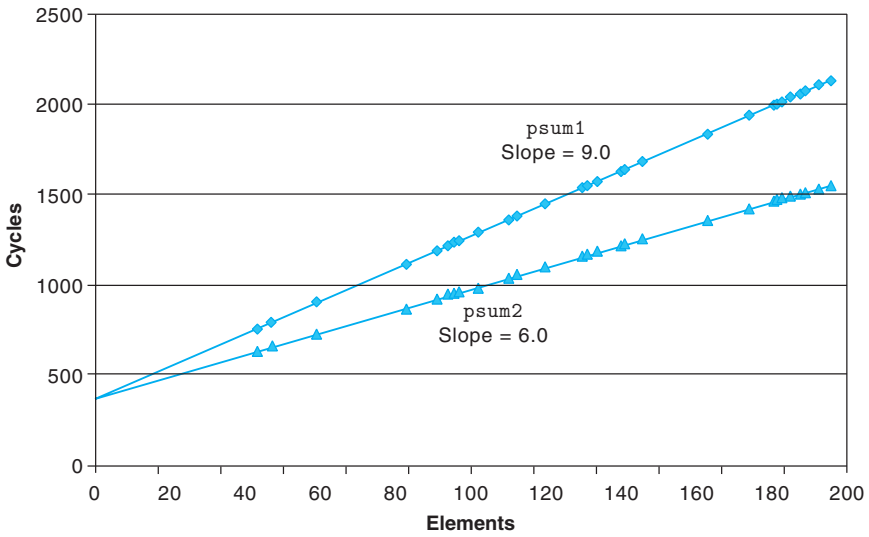



**Figure 5.2 Performance of prefix-sum functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

**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:

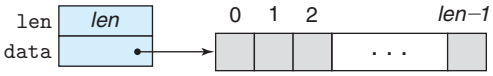



**Figure 5.3 Vector abstract data type.** A vector is represented by header information plus an array of designated length.

*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

```
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.

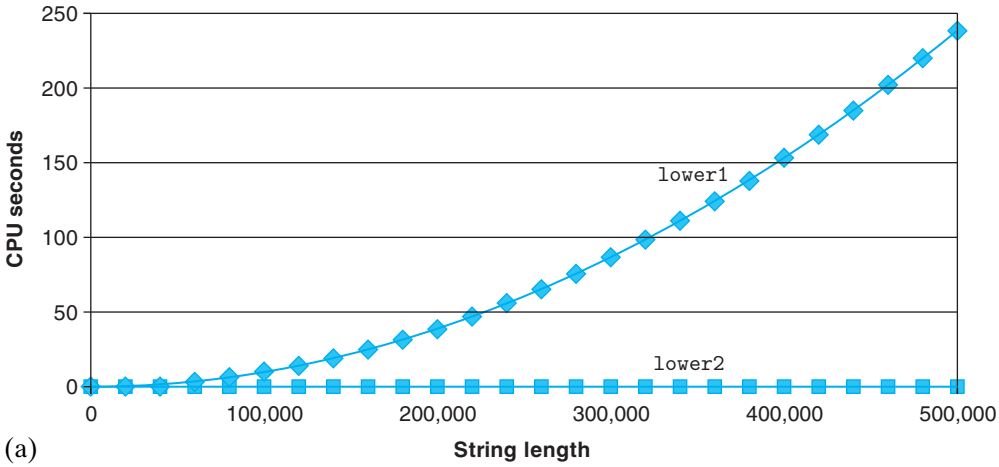
As an extreme example of the loop inefficiency seen in `combine1`, consider the procedure `lower1` shown in Figure 5.7. This procedure is styled after routines submitted by several students as part of a network programming project. Its purpose is to convert all of the uppercase letters in a string to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

*code/opt/vec.c*

```
1  data_t *get_vec_start(vec_ptr v)
2  {
3      return v->data;
4  }
```

*code/opt/vec.c*

```
1  /* Direct access to vector data */
2  void combine3(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      data_t *data = get_vec_start(v);
7
8      *dest = IDENT;
9      for (i = 0; i < length; i++) {
10         *dest = *dest OP data[i];
11     }
12 }
```

**Figure 5.9 Eliminating function calls within the loop.** The resulting code does not show a performance gain, but it enables additional optimizations.

Suppose instead that we add a function `get_vec_start` to our abstract data type. This function returns the starting address of the data array, as shown in Figure 5.9. We could then write the procedure shown as `combine3` in this figure, having no function calls in the inner loop. Rather than making a function call to retrieve each vector element, it accesses the array directly. A purist might say that this transformation seriously impairs the program modularity. In principle, the user of the vector abstract data type should not even need to know that the vector contents are stored as an array, rather than as some other data structure such as a linked list. A more pragmatic programmer would argue that this transformation is a necessary step toward achieving high-performance results.

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine2` | 545 | Move `vec_length` | 7.02 | 9.03 | 9.02 | 11.03 |
| `combine3` | 549 | Direct data access | 7.17 | 9.02 | 9.02 | 11.03 |

Surprisingly, there is no apparent performance improvement. Indeed, the performance for integer sum has gotten slightly worse. Evidently, other operations in the inner loop are forming a bottleneck that limits the performance more than the call to `get_vec_element`. We will return to this function later (Section 5.11.2) and see why the repeated bounds checking by `combine2` does not incur a performance penalty. For now, we can view this transformation as one of a series of steps that will ultimately lead to greatly improved performance.

## 5.6 Eliminating Unneeded Memory References

The code for `combine3` accumulates the value being computed by the combining operation at the location designated by the pointer `dest`. This attribute can be seen by examining the assembly code generated for the inner loop of the compiled code. We show here the x86-64 code generated for data type `double` and with multiplication as the combining operation:

```
    Inner loop of combine3.  data_t = double, OP = *
    dest in %rbx, data+i in %rdx, data+length in %rax
1   .L17:                                   loop:
2     vmovsd  (%rbx), %xmm0                   Read product from dest
3     vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
4     vmovsd  %xmm0, (%rbx)                   Store product at dest
5     addq    $8, %rdx                        Increment data+i
6     cmpq    %rax, %rdx                      Compare to data+length
7     jne     .L17                            If !=, goto loop
```

We see in this loop code that the address corresponding to pointer `dest` is held in register `%rbx`. It has also transformed the code to maintain a pointer to the *i*th data element in register `%rdx`, shown in the annotations as `data+i`. This pointer is incremented by 8 on every iteration. The loop termination is detected by comparing this pointer to one stored in register `%rax`. We can see that the accumulated value is read from and written to memory on each iteration. This reading and writing is wasteful, since the value read from `dest` at the beginning of each iteration should simply be the value written at the end of the previous iteration.

We can eliminate this needless reading and writing of memory by rewriting the code in the style of `combine4` in Figure 5.10. We introduce a temporary variable `acc` that is used in the loop to accumulate the computed value. The result is stored at `dest` only after the loop has been completed. As the assembly code that follows shows, the compiler can now use register `%xmm0` to hold the accumulated value. Compared to the loop in `combine3`, we have reduced the memory operations per iteration from two reads and one write to just a single read.

```
    Inner loop of combine4.  data_t = double, OP = *
    acc in %xmm0, data+i in %rdx, data+length in %rax
1   .L25:                                   loop:
2     vmulsd  (%rdx), %xmm0, %xmm0            Multiply acc by data[i]
3     addq    $8, %rdx                        Increment data+i
4     cmpq    %rax, %rdx                      Compare to data+length
5     jne     .L25                            If !=, goto loop
```

We see a significant improvement in program performance, as shown in the following table:

```
1   /* Accumulate result in local variable */
2   void combine4(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       for (i = 0; i < length; i++) {
10          acc = acc OP data[i];
11      }
12      *dest = acc;
13  }
```

**Figure 5.10 Accumulating result in temporary.** Holding the accumulated value in local variable `acc` (short for "accumulator") eliminates the need to retrieve it from memory and write back the updated value on every loop iteration.

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine3` | 549 | Direct data access | 7.17 | 9.02 | 9.02 | 11.03 |
| `combine4` | 551 | Accumulate in temporary | 1.27 | 3.01 | 3.01 | 5.01 |

All of our times improve by factors ranging from 2.2× to 5.7×, with the integer addition case dropping to just 1.27 clock cycles per element.

Again, one might think that a compiler should be able to automatically transform the `combine3` code shown in Figure 5.9 to accumulate the value in a register, as it does with the code for `combine4` shown in Figure 5.10. In fact, however, the two functions can have different behaviors due to memory aliasing. Consider, for example, the case of integer data with multiplication as the operation and 1 as the identity element. Let $v = [2, 3, 5]$ be a vector of three elements and consider the following two function calls:

```
combine3(v, get_vec_start(v) + 2);
combine4(v, get_vec_start(v) + 2);
```

That is, we create an alias between the last element of the vector and the destination for storing the result. The two functions would then execute as follows:

| Function | Initial | Before loop | `i` = 0 | `i` = 1 | `i` = 2 | Final |
|---|---|---|---|---|---|---|
| `combine3` | [2, 3, 5] | [2, 3, 1] | [2, 3, 2] | [2, 3, 6] | [2, 3, 36] | [2, 3, 36] |
| `combine4` | [2, 3, 5] | [2, 3, 5] | [2, 3, 5] | [2, 3, 5] | [2, 3, 5] | [2, 3, 30] |

As shown previously, `combine3` accumulates its result at the destination, which in this case is the final vector element. This value is therefore set first to 1, then to $2 \cdot 1 = 2$, and then to $3 \cdot 2 = 6$. On the last iteration, this value is then multiplied by itself to yield a final value of 36. For the case of `combine4`, the vector remains unchanged until the end, when the final element is set to the computed result $1 \cdot 2 \cdot 3 \cdot 5 = 30$.

Of course, our example showing the distinction between `combine3` and `combine4` is highly contrived. One could argue that the behavior of `combine4` more closely matches the intention of the function description. Unfortunately, a compiler cannot make a judgment about the conditions under which a function might be used and what the programmer's intentions might be. Instead, when given `combine3` to compile, the conservative approach is to keep reading and writing memory, even though this is less efficient.

### Practice Problem 5.4 (solution page 610)

When we use GCC to compile `combine3` with command-line option `-O2`, we get code with substantially better CPE performance than with `-O1`:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine3` | 549 | Compiled `-O1` | 7.17 | 9.02 | 9.02 | 11.03 |
| `combine3` | 549 | Compiled `-O2` | 1.60 | 3.01 | 3.01 | 5.01 |
| `combine4` | 551 | Accumulate in temporary | 1.27 | 3.01 | 3.01 | 5.01 |

We achieve performance comparable to that for `combine4`, except for the case of integer sum, but even it improves significantly. On examining the assembly code generated by the compiler, we find an interesting variant for the inner loop:

```
    Inner loop of combine3.  data_t = double, OP = *.  Compiled -O2
    dest in %rbx, data+i in %rdx, data+length in %rax
    Accumulated product in %xmm0
1   .L22:                                   loop:
2     vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
3     addq    $8, %rdx                        Increment data+i
4     cmpq    %rax, %rdx                      Compare to data+length
5     vmovsd  %xmm0, (%rbx)                   Store product at dest
6     jne     .L22                            If !=, goto loop
```

We can compare this to the version created with optimization level 1:

```
    Inner loop of combine3.  data_t = double, OP = *.  Compiled -O1
    dest in %rbx, data+i in %rdx, data+length in %rax
1   .L17:                                   loop:
2     vmovsd  (%rbx), %xmm0                   Read product from dest
3     vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
4     vmovsd  %xmm0, (%rbx)                   Store product at dest
```

```
5       addq    $8, %rdx                Increment data+i
6       cmpq    %rax, %rdx              Compare to data+length
7       jne     .L17                    If !=, goto loop
```

We see that, besides some reordering of instructions, the only difference is that the more optimized version does not contain the `vmovsd` implementing the read from the location designated by `dest` (line 2).

A. How does the role of register `%xmm0` differ in these two loops?

B. Will the more optimized version faithfully implement the C code of `combine3`, including when there is memory aliasing between `dest` and the vector data?

C. Either explain why this optimization preserves the desired behavior, or give an example where it would produce different results than the less optimized code.

---

With this final transformation, we reached a point where we require just 1.25–5 clock cycles for each element to be computed. This is a considerable improvement over the original 9–11 cycles when we first enabled optimization. We would now like to see just what factors are constraining the performance of our code and how we can improve things even further.

## 5.7 Understanding Modern Processors

Up to this point, we have applied optimizations that did not rely on any features of the target machine. They simply reduced the overhead of procedure calls and eliminated some of the critical "optimization blockers" that cause difficulties for optimizing compilers. As we seek to push the performance further, we must consider optimizations that exploit the *microarchitecture* of the processor—that is, the underlying system design by which a processor executes instructions. Getting every last bit of performance requires a detailed analysis of the program as well as code generation tuned for the target processor. Nonetheless, we can apply some basic optimizations that will yield an overall performance improvement on a large class of processors. The detailed performance results we report here may not hold for other machines, but the general principles of operation and optimization apply to a wide variety of machines.

To understand ways to improve performance, we require a basic understanding of the microarchitectures of modern processors. Due to the large number of transistors that can be integrated onto a single chip, modern microprocessors employ complex hardware that attempts to maximize program performance. One result is that their actual operation is far different from the view that is perceived by looking at machine-level programs. At the code level, it appears as if instructions are executed one at a time, where each instruction involves fetching values from registers or memory, performing an operation, and storing results back to a register or memory location. In the actual processor, a number of instructions

are evaluated simultaneously, a phenomenon referred to as *instruction-level parallelism*. In some designs, there can be 100 or more instructions "in flight." Elaborate mechanisms are employed to make sure the behavior of this parallel execution exactly captures the sequential semantic model required by the machine-level program. This is one of the remarkable feats of modern microprocessors: they employ complex and exotic microarchitectures, in which multiple instructions can be executed in parallel, while presenting an operational view of simple sequential instruction execution.

Although the detailed design of a modern microprocessor is well beyond the scope of this book, having a general idea of the principles by which they operate suffices to understand how they achieve instruction-level parallelism. We will find that two different lower bounds characterize the maximum performance of a program. The *latency bound* is encountered when a series of operations must be performed in strict sequence, because the result of one operation is required before the next one can begin. This bound can limit program performance when the data dependencies in the code limit the ability of the processor to exploit instruction-level parallelism. The *throughput bound* characterizes the raw computing capacity of the processor's functional units. This bound becomes the ultimate limit on program performance.

### 5.7.1 Overall Operation

Figure 5.11 shows a very simplified view of a modern microprocessor. Our hypothetical processor design is based loosely on the structure of recent Intel processors. These processors are described in the industry as being *superscalar*, which means they can perform multiple operations on every clock cycle and *out of order*, meaning that the order in which instructions execute need not correspond to their ordering in the machine-level program. The overall design has two main parts: the *instruction control unit* (ICU), which is responsible for reading a sequence of instructions from memory and generating from these a set of primitive operations to perform on program data, and the *execution unit* (EU), which then executes these operations. Compared to the simple *in-order* pipeline we studied in Chapter 4, out-of-order processors require far greater and more complex hardware, but they are better at achieving higher degrees of instruction-level parallelism.

The ICU reads the instructions from an *instruction cache*—a special high-speed memory containing the most recently accessed instructions. In general, the ICU fetches well ahead of the currently executing instructions, so that it has enough time to decode these and send operations down to the EU. One problem, however, is that when a program hits a branch,[1] there are two possible directions the program might go. The branch can be *taken*, with control passing to the branch target. Alternatively, the branch can be *not taken*, with control passing to the next

1. We use the term "branch" specifically to refer to conditional jump instructions. Other instructions that can transfer control to multiple destinations, such as procedure return and indirect jumps, provide similar challenges for the processor.

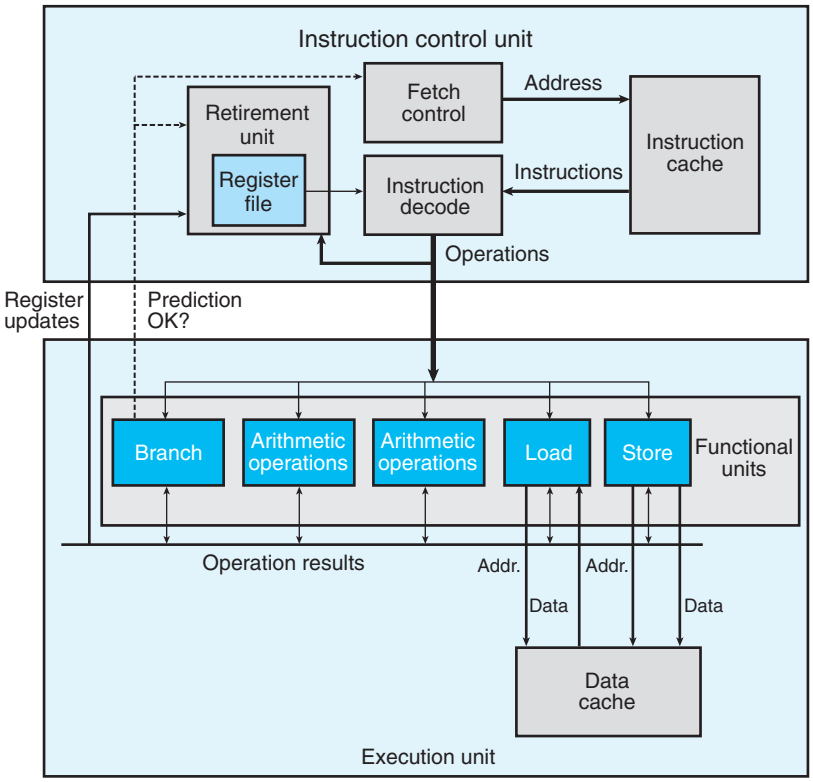



**Figure 5.11** **Block diagram of an out-of-order processor.** The instruction control unit is responsible for reading instructions from memory and generating a sequence of primitive operations. The execution unit then performs the operations and indicates whether the branches were correctly predicted.

instruction in the instruction sequence. Modern processors employ a technique known as *branch prediction*, in which they guess whether or not a branch will be taken and also predict the target address for the branch. Using a technique known as *speculative execution*, the processor begins fetching and decoding instructions at where it predicts the branch will go, and even begins executing these operations before it has been determined whether or not the branch prediction was correct. If it later determines that the branch was predicted incorrectly, it resets the state to that at the branch point and begins fetching and executing instructions in the other direction. The block labeled "Fetch control" incorporates branch prediction to perform the task of determining which instructions to fetch.

The *instruction decoding* logic takes the actual program instructions and converts them into a set of primitive *operations* (sometimes referred to as *micro-operations*). Each of these operations performs some simple computational task such as adding two numbers, reading data from memory, or writing data to memory. For machines with complex instructions, such as x86 processors, an instruction

can be decoded into multiple operations. The details of how instructions are decoded into sequences of operations varies between machines, and this information is considered highly proprietary. Fortunately, we can optimize our programs without knowing the low-level details of a particular machine implementation.

In a typical x86 implementation, an instruction that only operates on registers, such as

```
addq %rax,%rdx
```

is converted into a single operation. On the other hand, an instruction involving one or more memory references, such as

```
addq %rax,8(%rdx)
```

yields multiple operations, separating the memory references from the arithmetic operations. This particular instruction would be decoded as three operations: one to *load* a value from memory into the processor, one to add the loaded value to the value in register `%eax`, and one to *store* the result back to memory. The decoding splits instructions to allow a division of labor among a set of dedicated hardware units. These units can then execute the different parts of multiple instructions in parallel.

The EU receives operations from the instruction fetch unit. Typically, it can receive a number of them on each clock cycle. These operations are dispatched to a set of *functional units* that perform the actual operations. These functional units are specialized to handle different types of operations.

Reading and writing memory is implemented by the load and store units. The load unit handles operations that read data from the memory into the processor. This unit has an adder to perform address computations. Similarly, the store unit handles operations that write data from the processor to the memory. It also has an adder to perform address computations. As shown in the figure, the load and store units access memory via a *data cache*, a high-speed memory containing the most recently accessed data values.

With speculative execution, the operations are evaluated, but the final results are not stored in the program registers or data memory until the processor can be certain that these instructions should actually have been executed. Branch operations are sent to the EU, not to determine where the branch should go, but rather to determine whether or not they were predicted correctly. If the prediction was incorrect, the EU will discard the results that have been computed beyond the branch point. It will also signal the branch unit that the prediction was incorrect and indicate the correct branch destination. In this case, the branch unit begins fetching at the new location. As we saw in Section 3.6.6, such a *misprediction* incurs a significant cost in performance. It takes a while before the new instructions can be fetched, decoded, and sent to the functional units.

Figure 5.11 indicates that the different functional units are designed to perform different operations. Those labeled as performing "arithmetic operations" are typically specialized to perform different combinations of integer and floating-point operations. As the number of transistors that can be integrated onto a single

microprocessor chip has grown over time, successive models of microprocessors have increased the total number of functional units, the combinations of operations each unit can perform, and the performance of each of these units. The arithmetic units are intentionally designed to be able to perform a variety of different operations, since the required operations vary widely across different programs. For example, some programs might involve many integer operations, while others require many floating-point operations. If one functional unit were specialized to perform integer operations while another could only perform floating-point operations, then none of these programs would get the full benefit of having multiple functional units.

For example, our Intel Core i7 Haswell reference machine has eight functional units, numbered 0–7. Here is a partial list of each one's capabilities:

**0.** Integer arithmetic, floating-point multiplication, integer and floating-point division, branches

**1.** Integer arithmetic, floating-point addition, integer multiplication, floating-point multiplication

**2.** Load, address computation

**3.** Load, address computation

**4.** Store

**5.** Integer arithmetic

**6.** Integer arithmetic, branches

**7.** Store address computation

In the above list, "integer arithmetic" refers to basic operations, such as addition, bitwise operations, and shifting. Multiplication and division require more specialized resources. We see that a store operation requires two functional units—one to compute the store address and one to actually store the data. We will discuss the mechanics of store (and load) operations in Section 5.12.

We can see that this combination of functional units has the potential to perform multiple operations of the same type simultaneously. It has four units capable of performing integer operations, two that can perform load operations, and two that can perform floating-point multiplication. We will later see the impact these resources have on the maximum performance our programs can achieve.

Within the ICU, the *retirement unit* keeps track of the ongoing processing and makes sure that it obeys the sequential semantics of the machine-level program. Our figure shows a *register file* containing the integer, floating-point, and, more recently, SSE and AVX registers as part of the retirement unit, because this unit controls the updating of these registers. As an instruction is decoded, information about it is placed into a first-in, first-out queue. This information remains in the queue until one of two outcomes occurs. First, once the operations for the instruction have completed and any branch points leading to this instruction are confirmed as having been correctly predicted, the instruction can be *retired*, with any updates to the program registers being made. If some branch point leading to this instruction was mispredicted, on the other hand, the instruction will be

**Aside** The history of out-of-order processing

Out-of-order processing was first implemented in the Control Data Corporation 6600 processor in 1964. Instructions were processed by 10 different functional units, each of which could be operated independently. In its day, this machine, with a clock rate of 10 MHz, was considered the premium machine for scientific computing.

IBM first implemented out-of-order processing with the IBM 360/91 processor in 1966, but just to execute the floating-point instructions. For around 25 years, out-of-order processing was considered an exotic technology, found only in machines striving for the highest possible performance, until IBM reintroduced it in the RS/6000 line of workstations in 1990. This design became the basis for the IBM/Motorola PowerPC line, with the model 601, introduced in 1993, becoming the first single-chip microprocessor to use out-of-order processing. Intel introduced out-of-order processing with its PentiumPro model in 1995, with an underlying microarchitecture similar to that of our reference machine.

*flushed*, discarding any results that may have been computed. By this means, mispredictions will not alter the program state.

As we have described, any updates to the program registers occur only as instructions are being retired, and this takes place only after the processor can be certain that any branches leading to this instruction have been correctly predicted. To expedite the communication of results from one instruction to another, much of this information is exchanged among the execution units, shown in the figure as "Operation results." As the arrows in the figure show, the execution units can send results directly to each other. This is a more elaborate form of the data-forwarding techniques we incorporated into our simple processor design in Section 4.5.5.

The most common mechanism for controlling the communication of operands among the execution units is called *register renaming*. When an instruction that updates register $r$ is decoded, a *tag* $t$ is generated giving a unique identifier to the result of the operation. An entry $(r, t)$ is added to a table maintaining the association between program register $r$ and tag $t$ for an operation that will update this register. When a subsequent instruction using register $r$ as an operand is decoded, the operation sent to the execution unit will contain $t$ as the source for the operand value. When some execution unit completes the first operation, it generates a result $(v, t)$, indicating that the operation with tag $t$ produced value $v$. Any operation waiting for $t$ as a source will then use $v$ as the source value, a form of data forwarding. By this mechanism, values can be forwarded directly from one operation to another, rather than being written to and read from the register file, enabling the second operation to begin as soon as the first has completed. The renaming table only contains entries for registers having pending write operations. When a decoded instruction requires a register $r$, and there is no tag associated with this register, the operand is retrieved directly from the register file. With register renaming, an entire sequence of operations can be performed speculatively, even though the registers are updated only after the processor is certain of the branch outcomes.

| | Integer | | | Floating point | | |
|---|---|---|---|---|---|---|
| Operation | Latency | Issue | Capacity | Latency | Issue | Capacity |
| Addition | 1 | 1 | 4 | 3 | 1 | 1 |
| Multiplication | 3 | 1 | 1 | 5 | 1 | 2 |
| Division | 3–30 | 3–30 | 1 | 3–15 | 3–15 | 1 |

**Figure 5.12 Latency, issue time, and capacity characteristics of reference machine operations.** Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two independent operations. The capacity indicates how many of these operations can be issued simultaneously. The times for division depend on the data values.

### 5.7.2 Functional Unit Performance

Figure 5.12 documents the performance of some of the arithmetic operations for our Intel Core i7 Haswell reference machine, determined by both measurements and by reference to Intel literature [49]. These timings are typical for other processors as well. Each operation is characterized by its *latency*, meaning the total time required to perform the operation, the *issue time*, meaning the minimum number of clock cycles between two independent operations of the same type, and the *capacity*, indicating the number of functional units capable of performing that operation.

We see that the latencies increase in going from integer to floating-point operations. We see also that the addition and multiplication operations all have issue times of 1, meaning that on each clock cycle, the processor can start a new one of these operations. This short issue time is achieved through the use of *pipelining*. A pipelined function unit is implemented as a series of *stages*, each of which performs part of the operation. For example, a typical floating-point adder contains three stages (and hence the three-cycle latency): one to process the exponent values, one to add the fractions, and one to round the result. The arithmetic operations can proceed through the stages in close succession rather than waiting for one operation to complete before the next begins. This capability can be exploited only if there are successive, logically independent operations to be performed. Functional units with issue times of 1 cycle are said to be *fully pipelined:* they can start a new operation every clock cycle. Operations with capacity greater than 1 arise due to the capabilities of the multiple functional units, as was described earlier for the reference machine.

We see also that the divider (used for integer and floating-point division, as well as floating-point square root) is not pipelined—its issue time equals its latency. What this means is that the divider must perform a complete division before it can begin a new one. We also see that the latencies and issue times for division are given as ranges, because some combinations of dividend and divisor require more steps than others. The long latency and issue times of division make it a comparatively costly operation.

A more common way of expressing issue time is to specify the maximum *throughput* of the unit, defined as the reciprocal of the issue time. A fully pipelined functional unit has a maximum throughput of 1 operation per clock cycle, while units with higher issue times have lower maximum throughput. Having multiple functional units can increase throughput even further. For an operation with capacity $C$ and issue time $I$, the processor can potentially achieve a throughput of $C/I$ operations per clock cycle. For example, our reference machine is capable of performing floating-point multiplication operations at a rate of 2 per clock cycle. We will see how this capability can be exploited to increase program performance.

Circuit designers can create functional units with wide ranges of performance characteristics. Creating a unit with short latency or with pipelining requires more hardware, especially for more complex functions such as multiplication and floating-point operations. Since there is only a limited amount of space for these units on the microprocessor chip, CPU designers must carefully balance the number of functional units and their individual performance to achieve optimal overall performance. They evaluate many different benchmark programs and dedicate the most resources to the most critical operations. As Figure 5.12 indicates, integer multiplication and floating-point multiplication and addition were considered important operations in the design of the Core i7 Haswell processor, even though a significant amount of hardware is required to achieve the low latencies and high degree of pipelining shown. On the other hand, division is relatively infrequent and difficult to implement with either short latency or full pipelining.

The latencies, issue times, and capacities of these arithmetic operations can affect the performance of our combining functions. We can express these effects in terms of two fundamental bounds on the CPE values:

| | Integer | | Floating point | |
|---|---|---|---|---|
| Bound | + | * | + | * |
| Latency | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput | 0.50 | 1.00 | 1.00 | 0.50 |

The *latency bound* gives a minimum value for the CPE for any function that must perform the combining operation in a strict sequence. The *throughput bound* gives a minimum bound for the CPE based on the maximum rate at which the functional units can produce results. For example, since there is only one integer multiplier, and it has an issue time of 1 clock cycle, the processor cannot possibly sustain a rate of more than 1 multiplication per clock cycle. On the other hand, with four functional units capable of performing integer addition, the processor can potentially sustain a rate of 4 operations per cycle. Unfortunately, the need to read elements from memory creates an additional throughput bound. The two load units limit the processor to reading at most 2 data values per clock cycle, yielding a throughput bound of 0.50. We will demonstrate the effect of both the latency and throughput bounds with different versions of the combining functions.

### 5.7.3 An Abstract Model of Processor Operation

As a tool for analyzing the performance of a machine-level program executing on a modern processor, we will use a *data-flow* representation of programs, a graphical notation showing how the data dependencies between the different operations constrain the order in which they are executed. These constraints then lead to *critical paths* in the graph, putting a lower bound on the number of clock cycles required to execute a set of machine instructions.

Before proceeding with the technical details, it is instructive to examine the CPE measurements obtained for function `combine4`, our fastest code up to this point:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine4` | 551 | Accumulate in temporary | 1.27 | 3.01 | 3.01 | 5.01 |
| Latency bound | | | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | | | 0.50 | 1.00 | 1.00 | 0.50 |

We can see that these measurements match the latency bound for the processor, except for the case of integer addition. This is not a coincidence—it indicates that the performance of these functions is dictated by the latency of the sum or product computation being performed. Computing the product or sum of $n$ elements requires around $L \cdot n + K$ clock cycles, where $L$ is the latency of the combining operation and $K$ represents the overhead of calling the function and initiating and terminating the loop. The CPE is therefore equal to the latency bound $L$.

#### From Machine-Level Code to Data-Flow Graphs

Our data-flow representation of programs is informal. We use it as a way to visualize how the data dependencies in a program dictate its performance. We present the data-flow notation by working with `combine4` (Figure 5.10) as an example. We focus just on the computation performed by the loop, since this is the dominating factor in performance for large vectors. We consider the case of data type `double` with multiplication as the combining operation. Other combinations of data type and operation yield similar code. The compiled code for this loop consists of four instructions, with registers `%rdx` holding a pointer to the $i$th element of array `data`, `%rax` holding a pointer to the end of the array, and `%xmm0` holding the accumulated value `acc`.

```
    Inner loop of combine4.  data_t = double, OP = *
    acc in %xmm0, data+i in %rdx, data+length in %rax
1   .L25:                                  loop:
2     vmulsd  (%rdx), %xmm0, %xmm0           Multiply acc by data[i]
3     addq    $8, %rdx                       Increment data+i
4     cmpq    %rax, %rdx                     Compare to data+length
5     jne     .L25                           If !=, goto loop
```

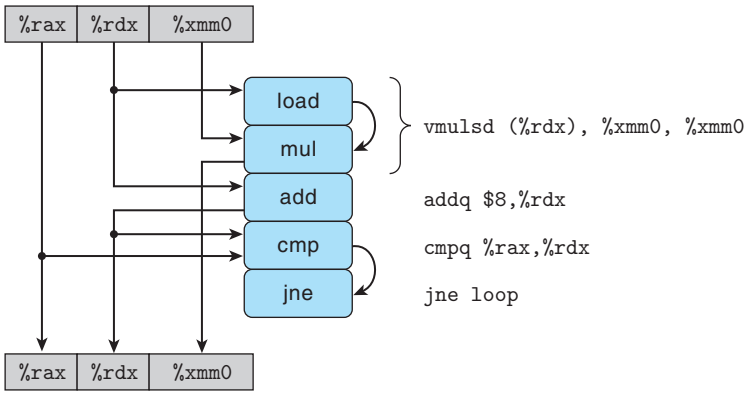



**Figure 5.13 Graphical representation of inner-loop code for `combine4`.** Instructions are dynamically translated into one or two operations, each of which receives values from other operations or from registers and produces values for other operations and for registers. We show the target of the final instruction as the label `loop`. It jumps to the first instruction shown.

As Figure 5.13 indicates, with our hypothetical processor design, the four instructions are expanded by the instruction decoder into a series of five *operations*, with the initial multiplication instruction being expanded into a load operation to read the source operand from memory, and a mul operation to perform the multiplication.

As a step toward generating a data-flow graph representation of the program, the boxes and lines along the left-hand side of Figure 5.13 show how the registers are used and updated by the different operations, with the boxes along the top representing the register values at the beginning of the loop, and those along the bottom representing the values at the end. For example, register `%rax` is only used as a source value by the cmp operation, and so the register has the same value at the end of the loop as at the beginning. Register `%rdx`, on the other hand, is both used and updated within the loop. Its initial value is used by the load and add operations; its new value is generated by the add operation, which is then used by the cmp operation. Register `%xmm0` is also updated within the loop by the mul operation, which first uses the initial value as a source value.

Some of the operations in Figure 5.13 produce values that do not correspond to registers. We show these as arcs between operations on the right-hand side. The load operation reads a value from memory and passes it directly to the mul operation. Since these two operations arise from decoding a single `vmulsd` instruction, there is no register associated with the intermediate value passing between them. The cmp operation updates the condition codes, and these are then tested by the jne operation.

For a code segment forming a loop, we can classify the registers that are accessed into four categories:

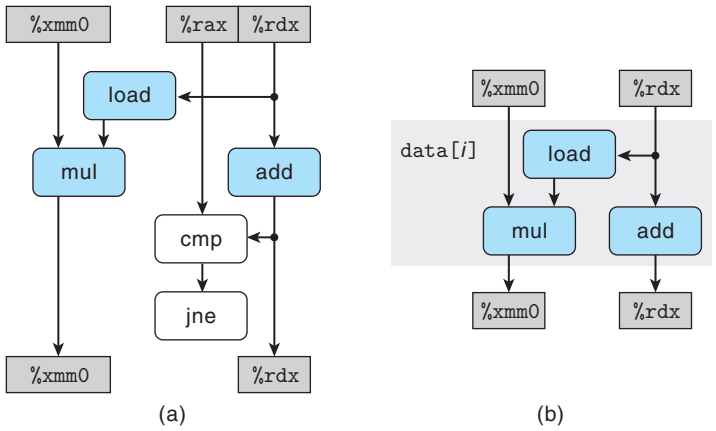



**Figure 5.14**
**Abstracting `combine4` operations as a data-flow graph.** We rearrange the operators of Figure 5.13 to more clearly show the data dependencies (a), and then further show only those operations that use values from one iteration to produce new values for the next (b).

*Read-only.* These are used as source values, either as data or to compute memory addresses, but they are not modified within the loop. The only read-only register for the loop in `combine4` is `%rax`.

*Write-only.* These are used as the destinations of data-movement operations. There are no such registers in this loop.

*Local.* These are updated and used within the loop, but there is no dependency from one iteration to another. The condition code registers are examples for this loop: they are updated by the cmp operation and used by the jne operation, but this dependency is contained within individual iterations.

*Loop.* These are used both as source values and as destinations for the loop, with the value generated in one iteration being used in another. We can see that `%rdx` and `%xmm0` are loop registers for `combine4`, corresponding to program values `data+i` and `acc`.

As we will see, the chains of operations between loop registers determine the performance-limiting data dependencies.

Figure 5.14 shows further refinements of the graphical representation of Figure 5.13, with a goal of showing only those operations and data dependencies that affect the program execution time. We see in Figure 5.14(a) that we rearranged the operators to show more clearly the flow of data from the source registers at the top (both read-only and loop registers) and to the destination registers at the bottom (both write-only and loop registers).

In Figure 5.14(a), we also color operators white if they are not part of some chain of dependencies between loop registers. For this example, the comparison (cmp) and branch (jne) operations do not directly affect the flow of data in the program. We assume that the instruction control unit predicts that branch will be taken, and hence the program will continue looping. The purpose of the compare and branch operations is to test the branch condition and notify the ICU if it is

not taken. We assume this checking can be done quickly enough that it does not slow down the processor.

In Figure 5.14(b), we have eliminated the operators that were colored white on the left, and we have retained only the loop registers. What we have left is an abstract template showing the data dependencies that form among loop registers due to one iteration of the loop. We can see in this diagram that there are two data dependencies from one iteration to the next. Along one side, we see the dependencies between successive values of program value `acc`, stored in register `%xmm0`. The loop computes a new value for `acc` by multiplying the old value by a data element, generated by the load operation. Along the other side, we see the dependencies between successive values of the pointer to the $i$th data element. On each iteration, the old value is used as the address for the load operation, and it is also incremented by the add operation to compute its new value.

Figure 5.15 shows the data-flow representation of $n$ iterations by the inner loop of function `combine4`. This graph was obtained by simply replicating the template shown in Figure 5.14(b) $n$ times. We can see that the program has two chains of data

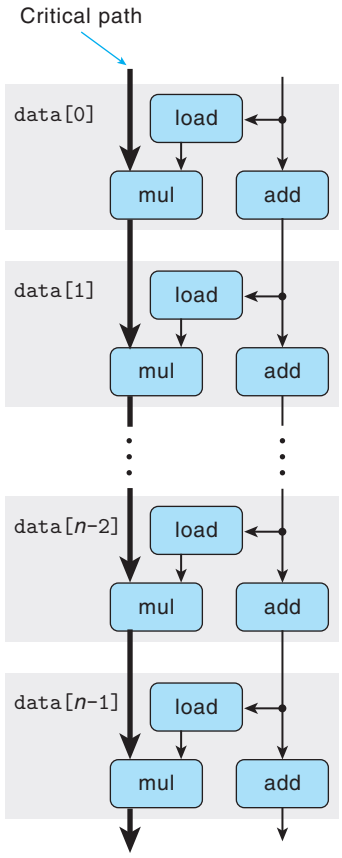



**Figure 5.15**
**Data-flow representation of computation by $n$ iterations of the inner loop of `combine4`.** The sequence of multiplication operations forms a critical path that limits program performance.

dependencies, corresponding to the updating of program values `acc` and `data+i` with operations mul and add, respectively. Given that floating-point multiplication has a latency of 5 cycles, while integer addition has a latency of 1 cycle, we can see that the chain on the left will form a *critical path*, requiring $5n$ cycles to execute. The chain on the right would require only $n$ cycles to execute, and so it does not limit the program performance.

Figure 5.15 demonstrates why we achieved a CPE equal to the latency bound of 5 cycles for `combine4`, when performing floating-point multiplication. When executing the function, the floating-point multiplier becomes the limiting resource. The other operations required during the loop—manipulating and testing pointer value `data+i` and reading data from memory—proceed in parallel with the multiplication. As each successive value of `acc` is computed, it is fed back around to compute the next value, but this will not occur until 5 cycles later.

The flow for other combinations of data type and operation are identical to those shown in Figure 5.15, but with a different data operation forming the chain of data dependencies shown on the left. For all of the cases where the operation has a latency $L$ greater than 1, we see that the measured CPE is simply $L$, indicating that this chain forms the performance-limiting critical path.

### Other Performance Factors

For the case of integer addition, on the other hand, our measurements of `combine4` show a CPE of 1.27, slower than the CPE of 1.00 we would predict based on the chains of dependencies formed along either the left- or the right-hand side of the graph of Figure 5.15. This illustrates the principle that the critical paths in a data-flow representation provide only a *lower* bound on how many cycles a program will require. Other factors can also limit performance, including the total number of functional units available and the number of data values that can be passed among the functional units on any given step. For the case of integer addition as the combining operation, the data operation is sufficiently fast that the rest of the operations cannot supply data fast enough. Determining exactly why the program requires 1.27 cycles per element would require a much more detailed knowledge of the hardware design than is publicly available.

To summarize our performance analysis of `combine4`: our abstract data-flow representation of program operation showed that `combine4` has a critical path of length $L \cdot n$ caused by the successive updating of program value `acc`, and this path limits the CPE to at least $L$. This is indeed the CPE we measure for all cases except integer addition, which has a measured CPE of 1.27 rather than the CPE of 1.00 we would expect from the critical path length.

It may seem that the latency bound forms a fundamental limit on how fast our combining operations can be performed. Our next task will be to restructure the operations to enhance instruction-level parallelism. We want to transform the program in such a way that our only limitation becomes the throughput bound, yielding CPEs below or close to 1.00.

### Practice Problem 5.5 (solution page 611)

Suppose we wish to write a function to evaluate a polynomial, where a polynomial of degree $n$ is defined to have a set of coefficients $a_0, a_1, a_2, \ldots, a_n$. For a value $x$, we evaluate the polynomial by computing

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{5.2}$$

This evaluation can be implemented by the following function, having as arguments an array of coefficients `a`, a value `x`, and the polynomial degree `degree` (the value $n$ in Equation 5.2). In this function, we compute both the successive terms of the equation and the successive powers of $x$ within a single loop:

```
1   double poly(double a[], double x, long degree)
2   {
3       long i;
4       double result = a[0];
5       double xpwr = x;  /* Equals x^i at start of loop */
6       for (i = 1; i <= degree; i++) {
7           result += a[i] * xpwr;
8           xpwr = x * xpwr;
9       }
10      return result;
11  }
```

A. For degree $n$, how many additions and how many multiplications does this code perform?

B. On our reference machine, with arithmetic operations having the latencies shown in Figure 5.12, we measure the CPE for this function to be 5.00. Explain how this CPE arises based on the data dependencies formed between iterations due to the operations implementing lines 7–8 of the function.

### Practice Problem 5.6 (solution page 611)

Let us continue exploring ways to evaluate polynomials, as described in Practice Problem 5.5. We can reduce the number of multiplications in evaluating a polynomial by applying *Horner's method*, named after British mathematician William G. Horner (1786–1837). The idea is to repeatedly factor out the powers of $x$ to get the following evaluation:

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)) \tag{5.3}$$

Using Horner's method, we can implement polynomial evaluation using the following code:

```
1   /* Apply Horner's method */
2   double polyh(double a[], double x, long degree)
3   {
```

```
4       long i;
5       double result = a[degree];
6       for (i = degree-1; i >= 0; i--)
7           result = a[i] + x*result;
8       return result;
9   }
```

A. For degree $n$, how many additions and how many multiplications does this code perform?

B. On our reference machine, with the arithmetic operations having the latencies shown in Figure 5.12, we measure the CPE for this function to be 8.00. Explain how this CPE arises based on the data dependencies formed between iterations due to the operations implementing line 7 of the function.

C. Explain how the function shown in Practice Problem 5.5 can run faster, even though it requires more operations.

## 5.8 Loop Unrolling

Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration. We saw an example of this with the function `psum2` (Figure 5.1), where each iteration computes two elements of the prefix sum, thereby halving the total number of iterations required. Loop unrolling can improve performance in two ways. First, it reduces the number of operations that do not contribute directly to the program result, such as loop indexing and conditional branching. Second, it exposes ways in which we can further transform the code to reduce the number of operations in the critical paths of the overall computation. In this section, we will examine simple loop unrolling, without any further transformations.

Figure 5.16 shows a version of our combining code using what we will refer to as "$2 \times 1$ loop unrolling." The first loop steps through the array two elements at a time. That is, the loop index `i` is incremented by 2 on each iteration, and the combining operation is applied to array elements $i$ and $i + 1$ in a single iteration.

In general, the vector length will not be a multiple of 2. We want our code to work correctly for arbitrary vector lengths. We account for this requirement in two ways. First, we make sure the first loop does not overrun the array bounds. For a vector of length $n$, we set the loop limit to be $n - 1$. We are then assured that the loop will only be executed when the loop index $i$ satisfies $i < n - 1$, and hence the maximum array index $i + 1$ will satisfy $i + 1 < (n - 1) + 1 = n$.

We can generalize this idea to unroll a loop by any factor $k$, yielding $k \times 1$ *loop unrolling*. To do so, we set the upper limit to be $n - k + 1$ and within the loop apply the combining operation to elements $i$ through $i + k - 1$. Loop index `i` is incremented by $k$ in each iteration. The maximum array index $i + k - 1$ will then be less than $n$. We include the second loop to step through the final few elements of the vector one at a time. The body of this loop will be executed between 0 and $k - 1$ times. For $k = 2$, we could use a simple conditional statement

```
1   /* 2 x 1 loop unrolling */
2   void combine5(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc = IDENT;
9
10      /* Combine 2 elements at a time */
11      for (i = 0; i < limit; i+=2) {
12          acc = (acc OP data[i]) OP data[i+1];
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          acc = acc OP data[i];
18      }
19      *dest = acc;
20  }
```

**Figure 5.16 Applying $2 \times 1$ loop unrolling.** This transformation can reduce the effect of loop overhead.

to optionally add a final iteration, as we did with the function `psum2` (Figure 5.1). For $k > 2$, the finishing cases are better expressed with a loop, and so we adopt this programming convention for $k = 2$ as well. We refer to this transformation as "$k \times 1$ loop unrolling," since we unroll by a factor of $k$ but accumulate values in a single variable `acc`.

### Practice Problem 5.7 (solution page 611)

Modify the code for `combine5` to unroll the loop by a factor $k = 5$.

When we measure the performance of unrolled code for unrolling factors $k = 2$ (`combine5`) and $k = 3$, we get the following results:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine4` | 551 | No unrolling | 1.27 | 3.01 | 3.01 | 5.01 |
| `combine5` | 568 | $2 \times 1$ unrolling | 1.01 | 3.01 | 3.01 | 5.01 |
| | | $3 \times 1$ unrolling | 1.01 | 3.01 | 3.01 | 5.01 |
| Latency bound | | | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | | | 0.50 | 1.00 | 1.00 | 0.50 |

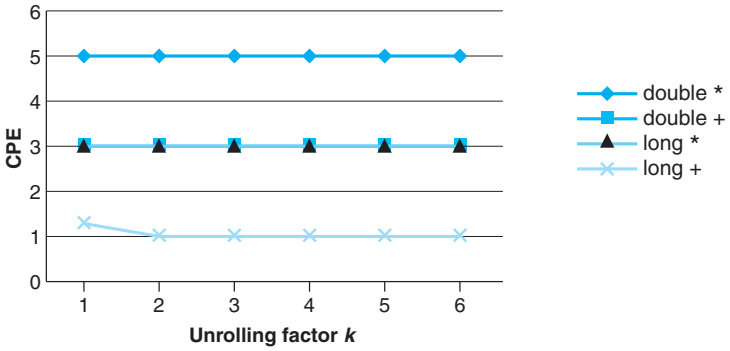



**Figure 5.17**
**CPE performance for different degrees of $k \times 1$ loop unrolling.** Only integer addition improves with this transformation.

We see that the CPE for integer addition improves, achieving the latency bound of 1.00. This result can be attributed to the benefits of reducing loop overhead operations. By reducing the number of overhead operations relative to the number of additions required to compute the vector sum, we can reach the point where the 1-cycle latency of integer addition becomes the performance-limiting factor. On the other hand, none of the other cases improve—they are already at their latency bounds. Figure 5.17 shows CPE measurements when unrolling the loop by up to a factor of 10. We see that the trends we observed for unrolling by 2 and 3 continue—none go below their latency bounds.

To understand why $k \times 1$ unrolling cannot improve performance beyond the latency bound, let us examine the machine-level code for the inner loop of `combine5`, having $k = 2$. The following code gets generated when type `data_t` is `double`, and the operation is multiplication:

```
    Inner loop of combine5.  data_t = double, OP = *
    i in %rdx, data %rax, limit in %rbx, acc in %xmm0
1   .L35:                                   loop:
2     vmulsd   (%rax,%rdx,8), %xmm0, %xmm0    Multiply acc by data[i]
3     vmulsd   8(%rax,%rdx,8), %xmm0, %xmm0   Multiply acc by data[i+1]
4     addq     $2, %rdx                       Increment i by 2
5     cmpq     %rdx, %rbp                     Compare to limit:i
6     jg       .L35                           If >, goto loop
```

We can see that GCC uses a more direct translation of the array referencing seen in the C code, compared to the pointer-based code generated for `combine4`.[2] Loop index `i` is held in register `%rdx`, and the address of `data` is held in register `%rax`. As before, the accumulated value `acc` is held in vector register `%xmm0`. The loop unrolling leads to two `vmulsd` instructions—one to add `data[i]` to `acc`, and

2. The GCC optimizer operates by generating multiple variants of a function and then choosing one that it predicts will yield the best performance and smallest code size. As a consequence, small changes in the source code can yield widely varying forms of machine code. We have found that the choice of pointer-based or array-based code has no impact on the performance of programs running on our reference machine.

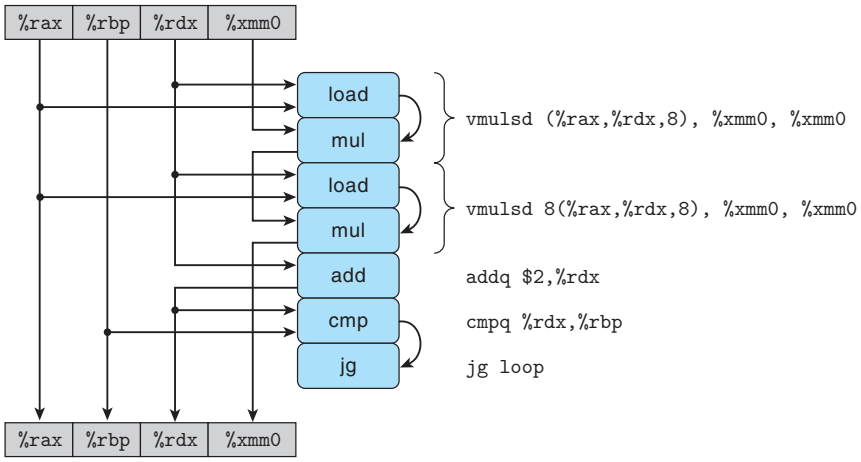
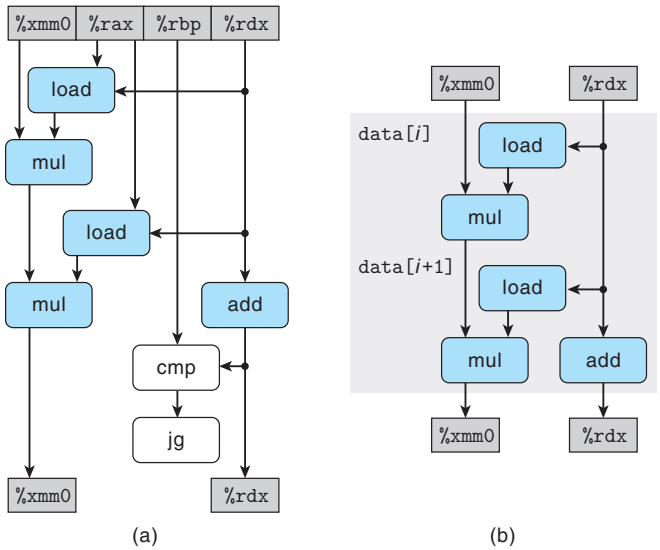



**Figure 5.18 Graphical representation of inner-loop code for `combine5`.** Each iteration has two `vmulsd` instructions, each of which is translated into a load and a mul operation.

**Figure 5.19**
**Abstracting `combine5` operations as a data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.18 to show the data dependencies between successive iterations (a). We see that each iteration must perform two multiplications in sequence (b).




the second to add `data[i+1]` to `acc`. Figure 5.18 shows a graphical representation of this code. The `vmulsd` instructions each get translated into two operations: one to load an array element from memory and one to multiply this value by the accumulated value. We see here that register `%xmm0` gets read and written twice in each execution of the loop. We can rearrange, simplify, and abstract this graph, following the process shown in Figure 5.19(a), to obtain the template shown in Figure 5.19(b). We then replicate this template $n/2$ times to show the computation for a vector of length $n$, obtaining the data-flow representation

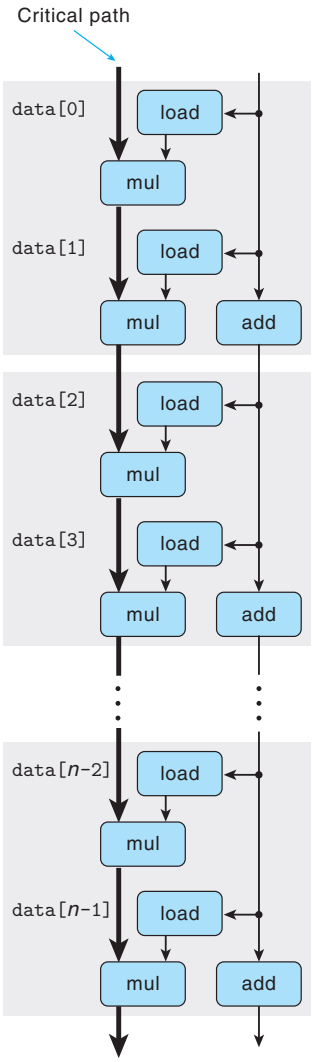



**Figure 5.20**
**Data-flow representation of `combine5` operating on a vector of length $n$.** Even though the loop has been unrolled by a factor of 2, there are still $n$ mul operations along the critical path.

shown in Figure 5.20. We see here that there is still a critical path of $n$ mul operations in this graph—there are half as many iterations, but each iteration has two multiplication operations in sequence. Since the critical path was the limiting factor for the performance of the code without loop unrolling, it remains so with $k \times 1$ loop unrolling.

**Aside** Getting the compiler to unroll loops

Loop unrolling can easily be performed by a compiler. Many compilers do this as part of their collection of optimizations. GCC will perform some forms of loop unrolling when invoked with optimization level 3 or higher.

## 5.9 Enhancing Parallelism

At this point, our functions have hit the bounds imposed by the latencies of the arithmetic units. As we have noted, however, the functional units performing addition and multiplication are all fully pipelined, meaning that they can start new operations every clock cycle, and some of the operations can be performed by multiple functional units. The hardware has the potential to perform multiplications and additions at a much higher rate, but our code cannot take advantage of this capability, even with loop unrolling, since we are accumulating the value as a single variable `acc`. We cannot compute a new value for `acc` until the preceding computation has completed. Even though the functional unit computing a new value for `acc` can start a new operation every clock cycle, it will only start one every $L$ cycles, where $L$ is the latency of the combining operation. We will now investigate ways to break this sequential dependency and get performance better than the latency bound.

### 5.9.1 Multiple Accumulators

For a combining operation that is associative and commutative, such as integer addition or multiplication, we can improve performance by splitting the set of combining operations into two or more parts and combining the results at the end. For example, let $P_n$ denote the product of elements $a_0, a_1, \ldots, a_{n-1}$:

$$P_n = \prod_{i=0}^{n-1} a_i$$

Assuming $n$ is even, we can also write this as $P_n = PE_n \times PO_n$, where $PE_n$ is the product of the elements with even indices, and $PO_n$ is the product of the elements with odd indices:

$$PE_n = \prod_{i=0}^{n/2-1} a_{2i}$$

$$PO_n = \prod_{i=0}^{n/2-1} a_{2i+1}$$

Figure 5.21 shows code that uses this method. It uses both two-way loop unrolling, to combine more elements per iteration, and two-way parallelism, accumulating elements with even indices in variable `acc0` and elements with odd indices in variable `acc1`. We therefore refer to this as "2 × 2 loop unrolling." As before, we include a second loop to accumulate any remaining array elements for the case where the vector length is not a multiple of 2. We then apply the combining operation to `acc0` and `acc1` to compute the final result.

Comparing loop unrolling alone to loop unrolling with two-way parallelism, we obtain the following performance:

```
1   /* 2 x 2 loop unrolling */
2   void combine6(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc0 = IDENT;
9       data_t acc1 = IDENT;
10
11      /* Combine 2 elements at a time */
12      for (i = 0; i < limit; i+=2) {
13          acc0 = acc0 OP data[i];
14          acc1 = acc1 OP data[i+1];
15      }
16
17      /* Finish any remaining elements */
18      for (; i < length; i++) {
19          acc0 = acc0 OP data[i];
20      }
21      *dest = acc0 OP acc1;
22  }
```

**Figure 5.21 Applying $2 \times 2$ loop unrolling.** By maintaining multiple accumulators, this approach can make better use of the multiple functional units and their pipelining capabilities.

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine4` | 551 | Accumulate in temporary | 1.27 | 3.01 | 3.01 | 5.01 |
| `combine5` | 568 | $2 \times 1$ unrolling | 1.01 | 3.01 | 3.01 | 5.01 |
| `combine6` | 573 | $2 \times 2$ unrolling | 0.81 | 1.51 | 1.51 | 2.51 |
| Latency bound | | | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | | | 0.50 | 1.00 | 1.00 | 0.50 |

We see that we have improved the performance for all cases, with integer product, floating-point addition, and floating-point multiplication improving by a factor of around 2, and integer addition improving somewhat as well. Most significantly, we have broken through the barrier imposed by the latency bound. The processor no longer needs to delay the start of one sum or product operation until the previous one has completed.

To understand the performance of `combine6`, we start with the code and operation sequence shown in Figure 5.22. We can derive a template showing the

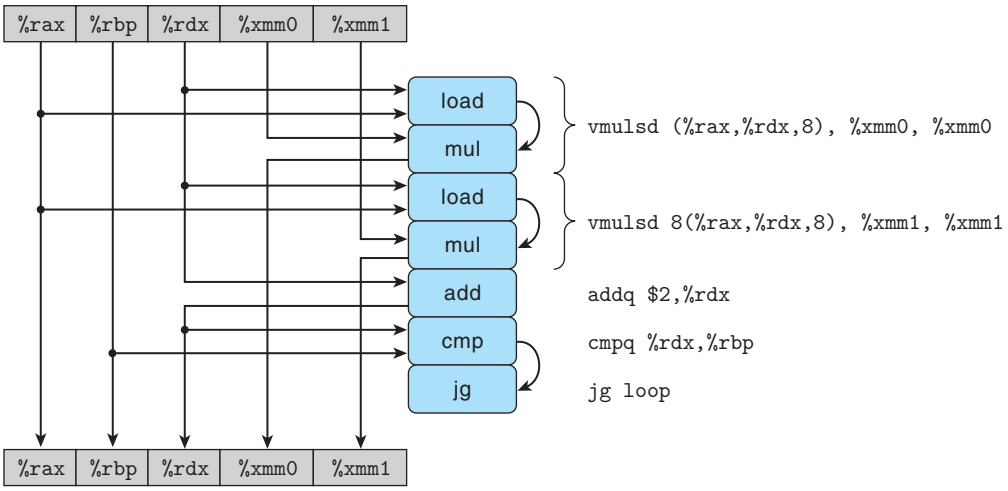



**Figure 5.22 Graphical representation of inner-loop code for `combine6`.** Each iteration has two `vmulsd` instructions, each of which is translated into a load and a mul operation.

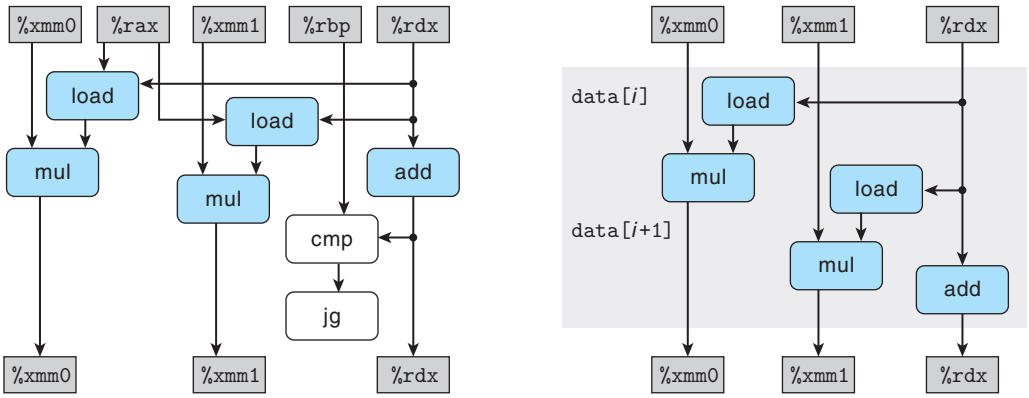



**Figure 5.23 Abstracting `combine6` operations as a data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.22 to show the data dependencies between successive iterations (a). We see that there is no dependency between the two mul operations (b).

data dependencies between iterations through the process shown in Figure 5.23. As with `combine5`, the inner loop contains two `vmulsd` operations, but these instructions translate into mul operations that read and write separate registers, with no data dependency between them (Figure 5.23(b)). We then replicate this template $n/2$ times (Figure 5.24), modeling the execution of the function on a vector of length $n$. We see that we now have two critical paths, one corresponding to computing the product of even-numbered elements (program value `acc0`) and

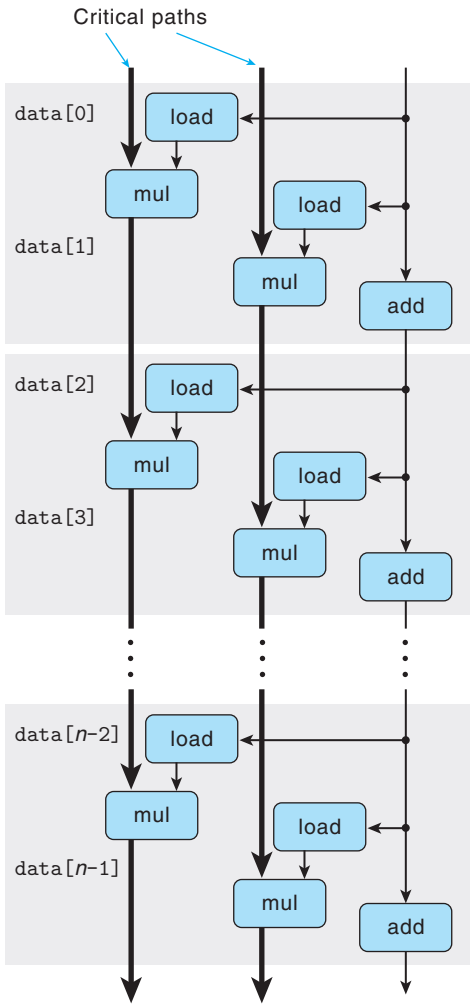



**Figure 5.24**
**Data-flow representation of `combine6` operating on a vector of length $n$.** We now have two critical paths, each containing $n/2$ operations.

one for the odd-numbered elements (program value `acc1`). Each of these critical paths contains only $n/2$ operations, thus leading to a CPE of around $5.00/2 = 2.50$. A similar analysis explains our observed CPE of around $L/2$ for operations with latency $L$ for the different combinations of data type and combining operation. Operationally, the programs are exploiting the capabilities of the functional units to increase their utilization by a factor of 2. The only exception is for integer addition. We have reduced the CPE to below 1.0, but there is still too much loop overhead to achieve the theoretical limit of 0.50.

We can generalize the multiple accumulator transformation to unroll the loop by a factor of $k$ and accumulate $k$ values in parallel, yielding $k \times k$ *loop unrolling*. Figure 5.25 demonstrates the effect of applying this transformation for values up to $k = 10$. We can see that, for sufficiently large values of $k$, the program can

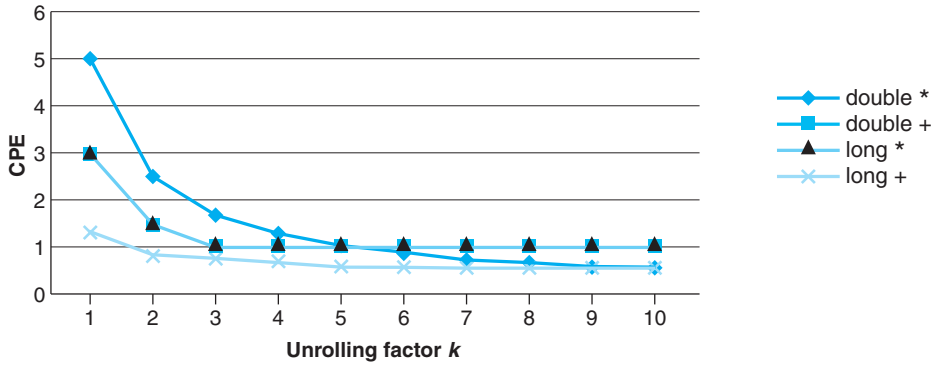



**Figure 5.25 CPE performance of $k \times k$ loop unrolling.** All of the CPEs improve with this transformation, achieving near or at their throughput bounds.

achieve nearly the throughput bounds for all cases. Integer addition achieves a CPE of 0.54 with $k = 7$, close to the throughput bound of 0.50 caused by the two load units. Integer multiplication and floating-point addition achieve CPEs of 1.01 when $k \geq 3$, approaching the throughput bound of 1.00 set by their functional units. Floating-point multiplication achieves a CPE of 0.51 for $k \geq 10$, approaching the throughput bound of 0.50 set by the two floating-point multipliers and the two load units. It is worth noting that our code is able to achieve nearly twice the throughput with floating-point multiplication as it can with floating-point addition, even though multiplication is a more complex operation.

In general, a program can achieve the throughput bound for an operation only when it can keep the pipelines filled for all of the functional units capable of performing that operation. For an operation with latency $L$ and capacity $C$, this requires an unrolling factor $k \geq C \cdot L$. For example, floating-point multiplication has $C = 2$ and $L = 5$, necessitating an unrolling factor of $k \geq 10$. Floating-point addition has $C = 1$ and $L = 3$, achieving maximum throughput with $k \geq 3$.

In performing the $k \times k$ unrolling transformation, we must consider whether it preserves the functionality of the original function. We have seen in Chapter 2 that two's-complement arithmetic is commutative and associative, even when overflow occurs. Hence, for an integer data type, the result computed by `combine6` will be identical to that computed by `combine5` under all possible conditions. Thus, an optimizing compiler could potentially convert the code shown in `combine4` first to a two-way unrolled variant of `combine5` by loop unrolling, and then to that of `combine6` by introducing parallelism. Some compilers do either this or similar transformations to improve performance for integer data.

On the other hand, floating-point multiplication and addition are not associative. Thus, `combine5` and `combine6` could produce different results due to rounding or overflow. Imagine, for example, a product computation in which all of the elements with even indices are numbers with very large absolute values, while those with odd indices are very close to 0.0. In such a case, product $PE_n$ might overflow, or $PO_n$ might underflow, even though computing product $P_n$ pro-

ceeds normally. In most real-life applications, however, such patterns are unlikely. Since most physical phenomena are continuous, numerical data tend to be reasonably smooth and well behaved. Even when there are discontinuities, they do not generally cause periodic patterns that lead to a condition such as that sketched earlier. It is unlikely that multiplying the elements in strict order gives fundamentally better accuracy than does multiplying two groups independently and then multiplying those products together. For most applications, achieving a performance gain of 2× outweighs the risk of generating different results for strange data patterns. Nevertheless, a program developer should check with potential users to see if there are particular conditions that may cause the revised algorithm to be unacceptable. Most compilers do not attempt such transformations with floating-point code, since they have no way to judge the risks of introducing transformations that can change the program behavior, no matter how small.

### 5.9.2 Reassociation Transformation

We now explore another way to break the sequential dependencies and thereby improve performance beyond the latency bound. We saw that the $k \times 1$ loop unrolling of `combine5` did not change the set of operations performed in combining the vector elements to form their sum or product. By a very small change in the code, however, we can fundamentally change the way the combining is performed, and also greatly increase the program performance.

Figure 5.26 shows a function `combine7` that differs from the unrolled code of `combine5` (Figure 5.16) only in the way the elements are combined in the inner loop. In `combine5`, the combining is performed by the statement

```
12      acc = (acc OP data[i]) OP data[i+1];
```

while in `combine7` it is performed by the statement

```
12      acc = acc OP (data[i] OP data[i+1]);
```

differing only in how two parentheses are placed. We call this a *reassociation transformation*, because the parentheses shift the order in which the vector elements are combined with the accumulated value `acc`, yielding a form of loop unrolling we refer to as "$2 \times 1a$."

To an untrained eye, the two statements may seem essentially the same, but when we measure the CPE, we get a surprising result:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine4` | 551 | Accumulate in temporary | 1.27 | 3.01 | 3.01 | 5.01 |
| `combine5` | 568 | $2 \times 1$ unrolling | 1.01 | 3.01 | 3.01 | 5.01 |
| `combine6` | 573 | $2 \times 2$ unrolling | 0.81 | 1.51 | 1.51 | 2.51 |
| `combine7` | 578 | $2 \times 1a$ unrolling | 1.01 | 1.51 | 1.51 | 2.51 |
| Latency bound | | | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | | | 0.50 | 1.00 | 1.00 | 0.50 |

```
1   /* 2 x 1a loop unrolling */
2   void combine7(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc = IDENT;
9
10      /* Combine 2 elements at a time */
11      for (i = 0; i < limit; i+=2) {
12          acc = acc OP (data[i] OP data[i+1]);
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          acc = acc OP data[i];
18      }
19      *dest = acc;
20  }
```

**Figure 5.26 Applying $2 \times 1a$ unrolling.** By reassociating the arithmetic, this approach increases the number of operations that can be performed in parallel.

The integer addition case matches the performance of $k \times 1$ unrolling (`combine5`), while the other three cases match the performance of the versions with parallel accumulators (`combine6`), doubling the performance relative to $k \times 1$ unrolling. These cases have broken through the barrier imposed by the latency bound.

Figure 5.27 illustrates how the code for the inner loop of `combine7` (for the case of multiplication as the combining operation and `double` as data type) gets decoded into operations and the resulting data dependencies. We see that the load operations resulting from the `vmovsd` and the first `vmulsd` instructions load vector elements $i$ and $i + 1$ from memory, and the first mul operation multiplies them together. The second mul operation then multiples this result by the accumulated value `acc`. Figure 5.28(a) shows how we rearrange, refine, and abstract the operations of Figure 5.27 to get a template representing the data dependencies for one iteration (Figure 5.28(b)). As with the templates for `combine5` and `combine7`, we have two load and two mul operations, but only one of the mul operations forms a data-dependency chain between loop registers. When we then replicate this template $n/2$ times to show the computations performed in multiplying $n$ vector elements (Figure 5.29), we see that we only have $n/2$ operations along the critical path. The first multiplication within each iteration can be performed without waiting for the accumulated value from the previous iteration. Thus, we reduce the minimum possible CPE by a factor of around 2.

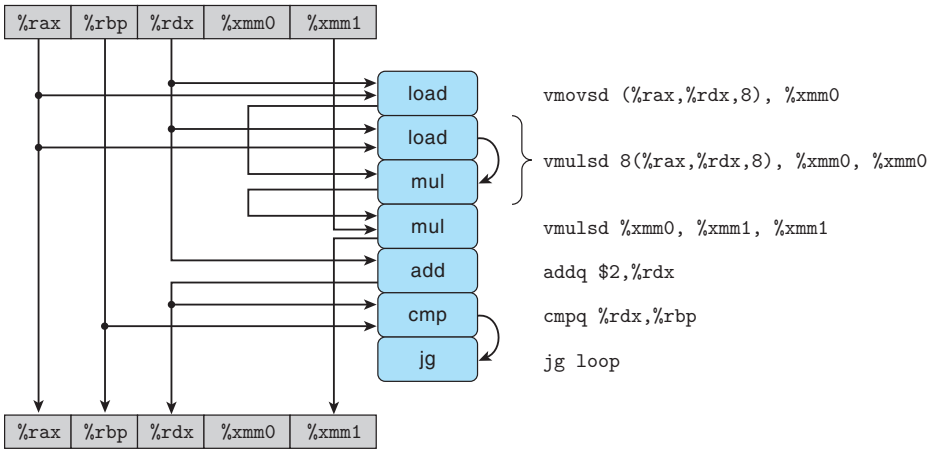



**Figure 5.27 Graphical representation of inner-loop code for `combine7`.** Each iteration gets decoded into similar operations as for `combine5` or `combine6`, but with different data dependencies.

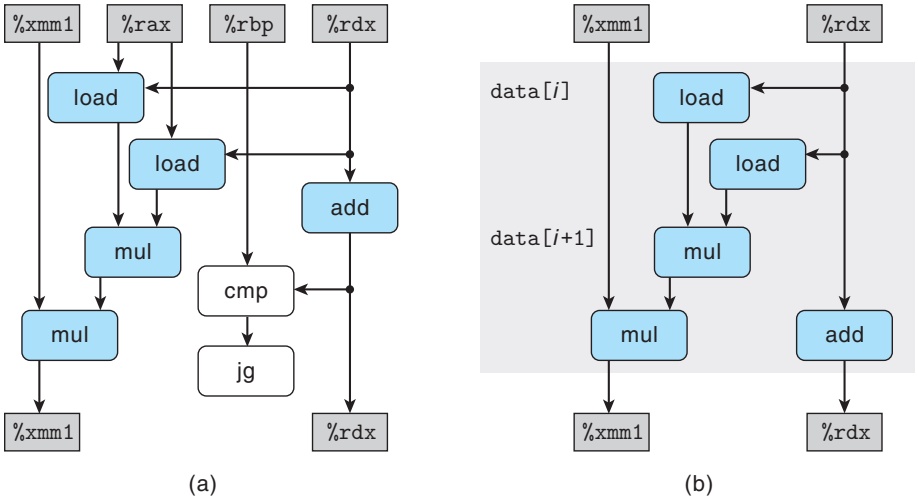



**Figure 5.28 Abstracting `combine7` operations as a data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.27 to show the data dependencies between successive iterations. The upper mul operation multiplies two 2-vector elements with each other, while the lower one multiplies the result by loop variable `acc`.

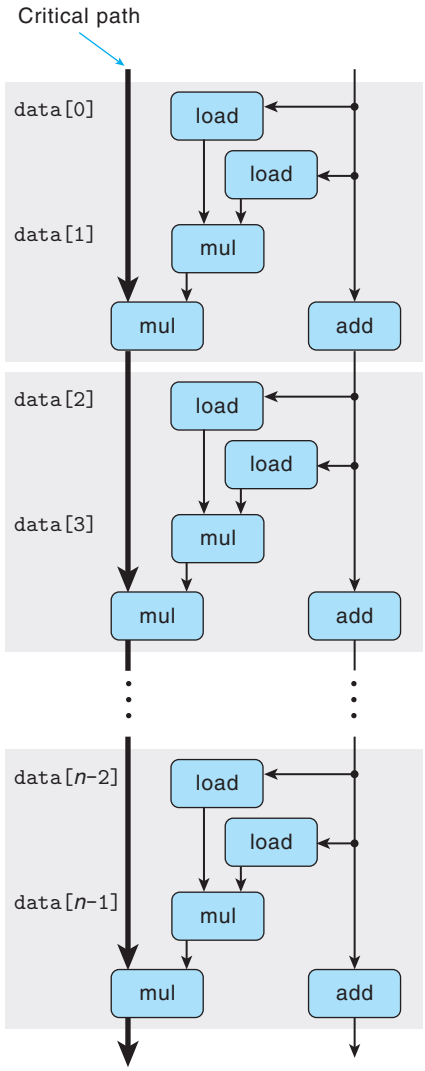



**Figure 5.29**
**Data-flow representation of `combine7` operating on a vector of length $n$.** We have a single critical path, but it contains only $n/2$ operations.

Figure 5.30 demonstrates the effect of applying the reassociation transformation to achieve what we refer to as $k \times 1a$ *loop unrolling* for values up to $k = 10$. We can see that this transformation yields performance results similar to what is achieved by maintaining $k$ separate accumulators with $k \times k$ unrolling. In all cases, we come close to the throughput bounds imposed by the functional units.

In performing the reassociation transformation, we once again change the order in which the vector elements will be combined together. For integer addition and multiplication, the fact that these operations are associative implies that this reordering will have no effect on the result. For the floating-point cases, we must once again assess whether this reassociation is likely to significantly affect

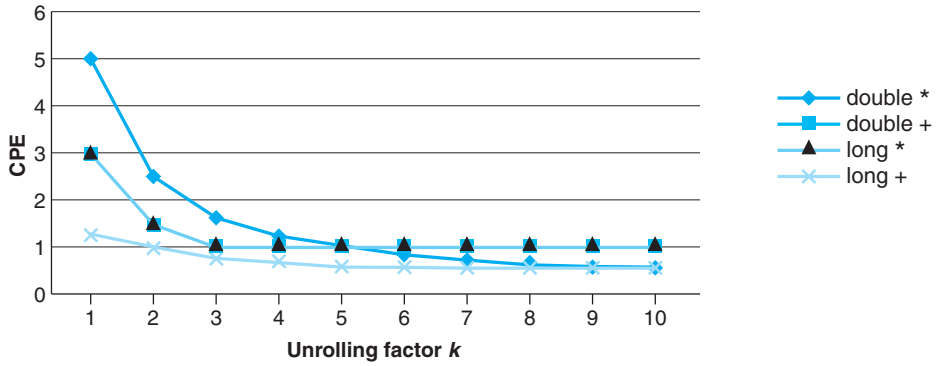



**Figure 5.30 CPE performance for $k \times 1a$ loop unrolling.** All of the CPEs improve with this transformation, nearly approaching their throughput bounds.

the outcome. We would argue that the difference would be immaterial for most applications.

In summary, a reassociation transformation can reduce the number of operations along the critical path in a computation, resulting in better performance by better utilizing the multiple functional units and their pipelining capabilities. Most compilers will not attempt any reassociations of floating-point operations, since these operations are not guaranteed to be associative. Current versions of GCC do perform reassociations of integer operations, but not always with good effects. In general, we have found that unrolling a loop and accumulating multiple values in parallel is a more reliable way to achieve improved program performance.

## Practice Problem 5.8 (solution page 612)

Consider the following function for computing the product of an array of $n$ double-precision numbers. We have unrolled the loop by a factor of 3.

```
double aprod(double a[], long n)
{
    long i;
    double x, y, z;
    double r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; /* Product computation */
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

For the line labeled "`Product computation`," we can use parentheses to create five different associations of the computation, as follows:

```
r = ((r * x) * y) * z; /* A1 */
r = (r * (x * y)) * z; /* A2 */
r = r * ((x * y) * z); /* A3 */
r = r * (x * (y * z)); /* A4 */
r = (r * x) * (y * z); /* A5 */
```

Assume we run these functions on a machine where floating-point multiplication has a latency of 5 clock cycles. Determine the lower bound on the CPE set by the data dependencies of the multiplication. (*Hint:* It helps to draw a data-flow representation of how `r` is computed on every iteration.)

**Web Aside OPT:SIMD** Achieving greater parallelism with vector instructions

As described in Section 3.1, Intel introduced the SSE instructions in 1999, where SSE is the acronym for "streaming SIMD extensions" and, in turn, SIMD (pronounced "sim-dee") is the acronym for "single instruction, multiple data." The SSE capability has gone through multiple generations, with more recent versions being named *advanced vector extensions*, or AVX. The SIMD execution model involves operating on entire vectors of data within single instructions. These vectors are held in a special set of *vector registers*, named `%ymm0`–`%ymm15`. Current AVX vector registers are 32 bytes long, and therefore each can hold eight 32-bit numbers or four 64-bit numbers, where the numbers can be either integer or floating-point values. AVX instructions can then perform vector operations on these registers, such as adding or multiplying eight or four sets of values in parallel. For example, if YMM register `%ymm0` contains eight single-precision floating-point numbers, which we denote $a_0, \ldots, a_7$, and `%rcx` contains the memory address of a sequence of eight single-precision floating-point numbers, which we denote $b_0, \ldots, b_7$, then the instruction

```
vmulps  (%rcs), %ymm0, %ymm1
```

will read the eight values from memory and perform eight multiplications in parallel, computing $a_i \leftarrow a_i \cdot b_i$, for $0 \le i \le 7$ and storing the resulting eight products in vector register `%ymm1`. We see that a single instruction is able to generate a computation over multiple data values, hence the term "SIMD."

GCC supports extensions to the C language that let programmers express a program in terms of vector operations that can be compiled into the vector instructions of AVX (as well as code based on the earlier SSE instructions). This coding style is preferable to writing code directly in assembly language, since GCC can also generate code for the vector instructions found on other processors.

Using a combination of GCC instructions, loop unrolling, and multiple accumulators, we are able to achieve the following performance for our combining functions:

**Web Aside OPT:SIMD** Achieving greater parallelism with vector instructions *(continued)*

| | Integer | | | | Floating point | | | |
|---|---|---|---|---|---|---|---|---|
| | `int` | | `long` | | `int` | | `long` | |
| Method | + | * | + | * | + | * | + | * |
| Scalar 10 × 10 | 0.54 | 1.01 | 0.55 | 1.00 | 1.01 | 0.51 | 1.01 | 0.52 |
| Scalar throughput bound | 0.50 | 0.50 | 1.00 | 1.00 | 1.00 | 1.00 | 0.50 | 0.50 |
| Vector 8 × 8 | 0.05 | 0.24 | 0.13 | 1.51 | 0.12 | 0.08 | 0.25 | 0.16 |
| Vector throughput bound | 0.06 | 0.12 | 0.12 | — | 0.12 | 0.06 | 0.25 | 0.12 |

In this chart, the first set of numbers is for conventional, *scalar* code written in the style of `combine6`, unrolling by a factor of 10 and maintaining 10 accumulators. The second set of numbers is for code written in a form that GCC can compile into AVX vector code. In addition to using vector operations, this version unrolls the main loop by a factor of 8 and maintains eight separate vector accumulators. We show results for both 32-bit and 64-bit numbers, since the vector instructions achieve 8-way parallelism in the first case, but only 4-way parallelism in the second.

We can see that the vector code achieves almost an eightfold improvement on the four 32-bit cases, and a fourfold improvement on three of the four 64-bit cases. Only the long integer multiplication code does not perform well when we attempt to express it in vector code. The AVX instruction set does not include one to do parallel multiplication of 64-bit integers, and so GCC cannot generate vector code for this case. Using vector instructions creates a new throughput bound for the combining operations. These are eight times lower for 32-bit operations and four times lower for 64-bit operations than the scalar limits. Our code comes close to achieving these bounds for several combinations of data type and operation.

## 5.10 Summary of Results for Optimizing Combining Code

Our efforts at maximizing the performance of a routine that adds or multiplies the elements of a vector have clearly paid off. The following summarizes the results we obtain with *scalar* code, not making use of the vector parallelism provided by AVX vector instructions:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine1` | 543 | Abstract `-O1` | 10.12 | 10.12 | 10.17 | 11.14 |
| `combine6` | 573 | 2 × 2 unrolling | 0.81 | 1.51 | 1.51 | 2.51 |
| | | 10 × 10 unrolling | 0.55 | 1.00 | 1.01 | 0.52 |
| Latency bound | | | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | | | 0.50 | 1.00 | 1.00 | 0.50 |

By using multiple optimizations, we have been able to achieve CPEs close to the throughput bounds of 0.50 and 1.00, limited only by the capacities of the functional units. These represent 10–20× improvements on the original code. This has all been done using ordinary C code and a standard compiler. Rewriting the code to take advantage of the newer SIMD instructions yields additional performance gains of nearly 4× or 8×. For example, for single-precision multiplication, the CPE drops from the original value of 11.14 down to 0.06, an overall performance gain of over 180×. This example demonstrates that modern processors have considerable amounts of computing power, but we may need to coax this power out of them by writing our programs in very stylized ways.

## 5.11 Some Limiting Factors

We have seen that the critical path in a data-flow graph representation of a program indicates a fundamental lower bound on the time required to execute a program. That is, if there is some chain of data dependencies in a program where the sum of all of the latencies along that chain equals $T$, then the program will require at least $T$ cycles to execute.

We have also seen that the throughput bounds of the functional units also impose a lower bound on the execution time for a program. That is, assume that a program requires a total of $N$ computations of some operation, that the microprocessor has $C$ functional units capable of performing that operation, and that these units have an issue time of $I$. Then the program will require at least $N \cdot I/C$ cycles to execute.

In this section, we will consider some other factors that limit the performance of programs on actual machines.

### 5.11.1 Register Spilling

The benefits of loop parallelism are limited by the ability to express the computation in assembly code. If a program has a degree of parallelism $P$ that exceeds the number of available registers, then the compiler will resort to *spilling*, storing some of the temporary values in memory, typically by allocating space on the run-time stack. As an example, the following measurements compare the result of extending the multiple accumulator scheme of `combine6` to the cases of $k = 10$ and $k = 20$:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine6` | 573 | | | | | |
| | | 10 × 10 unrolling | 0.55 | 1.00 | 1.01 | 0.52 |
| | | 20 × 20 unrolling | 0.83 | 1.03 | 1.02 | 0.68 |
| Throughput bound | | | 0.50 | 1.00 | 1.00 | 0.50 |

We can see that none of the CPEs improve with this increased unrolling, and some even get worse. Modern x86-64 processors have 16 integer registers and can make use of the 16 YMM registers to store floating-point data. Once the number of loop variables exceeds the number of available registers, the program must allocate some on the stack.

As an example, the following snippet of code shows how accumulator `acc0` is updated in the inner loop of the code with $10 \times 10$ unrolling:

```
Updating of accumulator acc0 in 10 x 10 urolling
vmulsd  (%rdx), %xmm0, %xmm0     acc0 *= data[i]
```

We can see that the accumulator is kept in register `%xmm0`, and so the program can simply read `data[i]` from memory and multiply it by this register.

The comparable part of the code for $20 \times 20$ unrolling has a much different form:

```
Updating of accumulator acc0 in 20 x 20 unrolling
vmovsd  40(%rsp), %xmm0
vmulsd  (%rdx), %xmm0, %xmm0
vmovsd  %xmm0, 40(%rsp)
```

The accumulator is kept as a local variable on the stack, at offset 40 from the stack pointer. The program must read both its value and the value of `data[i]` from memory, multiply them, and store the result back to memory.

Once a compiler must resort to register spilling, any advantage of maintaining multiple accumulators will most likely be lost. Fortunately, x86-64 has enough registers that most loops will become throughput limited before this occurs.

### 5.11.2 Branch Prediction and Misprediction Penalties

We demonstrated via experiments in Section 3.6.6 that a conditional branch can incur a significant *misprediction penalty* when the branch prediction logic does not correctly anticipate whether or not a branch will be taken. Now that we have learned something about how processors operate, we can understand where this penalty arises.

Modern processors work well ahead of the currently executing instructions, reading new instructions from memory and decoding them to determine what operations to perform on what operands. This *instruction pipelining* works well as long as the instructions follow in a simple sequence. When a branch is encountered, the processor must guess which way the branch will go. For the case of a conditional jump, this means predicting whether or not the branch will be taken. For an instruction such as an indirect jump (as we saw in the code to jump to an address specified by a jump table entry) or a procedure return, this means predicting the target address. In this discussion, we focus on conditional branches.

In a processor that employs *speculative execution*, the processor begins executing the instructions at the predicted branch target. It does this in a way that avoids modifying any actual register or memory locations until the actual outcome has been determined. If the prediction is correct, the processor can then

"commit" the results of the speculatively executed instructions by storing them in registers or memory. If the prediction is incorrect, the processor must discard all of the speculatively executed results and restart the instruction fetch process at the correct location. The misprediction penalty is incurred in doing this, because the instruction pipeline must be refilled before useful results are generated.

We saw in Section 3.6.6 that recent versions of x86 processors, including all processors capable of executing x86-64 programs, have *conditional move* instructions. GCC can generate code that uses these instructions when compiling conditional statements and expressions, rather than the more traditional realizations based on conditional transfers of control. The basic idea for translating into conditional moves is to compute the values along both branches of a conditional expression or statement and then use conditional moves to select the desired value. We saw in Section 4.5.7 that conditional move instructions can be implemented as part of the pipelined processing of ordinary instructions. There is no need to guess whether or not the condition will hold, and hence no penalty for guessing incorrectly.

How, then, can a C programmer make sure that branch misprediction penalties do not hamper a program's efficiency? Given the 19-cycle misprediction penalty we measured for the reference machine, the stakes are very high. There is no simple answer to this question, but the following general principles apply.

### Do Not Be Overly Concerned about Predictable Branches

We have seen that the effect of a mispredicted branch can be very high, but that does not mean that all program branches will slow a program down. In fact, the branch prediction logic found in modern processors is very good at discerning regular patterns and long-term trends for the different branch instructions. For example, the loop-closing branches in our combining routines would typically be predicted as being taken, and hence would only incur a misprediction penalty on the last time around.

As another example, consider the results we observed when shifting from `combine2` to `combine3`, when we took the function `get_vec_element` out of the inner loop of the function, as is reproduced below:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine2` | 545 | Move `vec_length` | 7.02 | 9.03 | 9.02 | 11.03 |
| `combine3` | 549 | Direct data access | 7.17 | 9.02 | 9.02 | 11.03 |

The CPE did not improve, even though the transformation eliminated two conditionals on each iteration that check whether the vector index is within bounds. For this function, the checks always succeed, and hence they are highly predictable.

As a way to measure the performance impact of bounds checking, consider the following combining code, where we have modified the inner loop of `combine4` by replacing the access to the data element with the result of performing an inline substitution of the code for `get_vec_element`. We will call this new version

`combine4b`. This code performs bounds checking and also references the vector elements through the vector data structure.

```
1   /* Include bounds check in loop */
2   void combine4b(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       data_t acc = IDENT;
7
8       for (i = 0; i < length; i++) {
9           if (i >= 0 && i < v->len) {
10              acc = acc OP v->data[i];
11          }
12      }
13      *dest = acc;
14  }
```

We can then directly compare the CPE for the functions with and without bounds checking:

| | | | Integer | | Floating point | |
|---|---|---|---|---|---|---|
| Function | Page | Method | + | * | + | * |
| `combine4` | 551 | No bounds checking | 1.27 | 3.01 | 3.01 | 5.01 |
| `combine4b` | 551 | Bounds checking | 2.02 | 3.01 | 3.01 | 5.01 |

The version with bounds checking is slightly slower for the case of integer addition, but it achieves the same performance for the other three cases. The performance of these cases is limited by the latencies of their respective combining operations. The additional computation required to perform bounds checking can take place in parallel with the combining operations. The processor is able to predict the outcomes of these branches, and so none of this evaluation has much effect on the fetching and processing of the instructions that form the critical path in the program execution.

## Write Code Suitable for Implementation with Conditional Moves

Branch prediction is only reliable for regular patterns. Many tests in a program are completely unpredictable, dependent on arbitrary features of the data, such as whether a number is negative or positive. For these, the branch prediction logic will do very poorly. For inherently unpredictable cases, program performance can be greatly enhanced if the compiler is able to generate code using conditional data transfers rather than conditional control transfers. This cannot be controlled directly by the C programmer, but some ways of expressing conditional behavior can be more directly translated into conditional moves than others.

We have found that GCC is able to generate conditional moves for code written in a more "functional" style, where we use conditional operations to compute

values and then update the program state with these values, as opposed to a more "imperative" style, where we use conditionals to selectively update program state.

There are no strict rules for these two styles, and so we illustrate with an example. Suppose we are given two arrays of integers `a` and `b`, and at each position $i$, we want to set `a[`$i$`]` to the minimum of `a[`$i$`]` and `b[`$i$`]`, and `b[`$i$`]` to the maximum.

An imperative style of implementing this function is to check at each position $i$ and swap the two elements if they are out of order:

```
1   /* Rearrange two vectors so that for each i, b[i] >= a[i] */
2   void minmax1(long a[], long b[], long n) {
3       long i;
4       for (i = 0; i < n; i++) {
5           if (a[i] > b[i]) {
6               long t = a[i];
7               a[i] = b[i];
8               b[i] = t;
9           }
10      }
11  }
```

Our measurements for this function show a CPE of around 13.5 for random data and 2.5–3.5 for predictable data, an indication of a misprediction penalty of around 20 cycles.

A functional style of implementing this function is to compute the minimum and maximum values at each position $i$ and then assign these values to `a[`$i$`]` and `b[`$i$`]`, respectively:

```
1   /* Rearrange two vectors so that for each i, b[i] >= a[i] */
2   void minmax2(long a[], long b[], long n) {
3       long i;
4       for (i = 0; i < n; i++) {
5           long min = a[i] < b[i] ? a[i] : b[i];
6           long max = a[i] < b[i] ? b[i] : a[i];
7           a[i] = min;
8           b[i] = max;
9       }
10  }
```

Our measurements for this function show a CPE of around 4.0 regardless of whether the data are arbitrary or predictable. (We also examined the generated assembly code to make sure that it indeed uses conditional moves.)

As discussed in Section 3.6.6, not all conditional behavior can be implemented with conditional data transfers, and so there are inevitably cases where programmers cannot avoid writing code that will lead to conditional branches for which the processor will do poorly with its branch prediction. But, as we have shown, a little cleverness on the part of the programmer can sometimes make code more amenable to translation into conditional data transfers. This requires some amount

of experimentation, writing different versions of the function and then examining the generated assembly code and measuring performance.

**Practice Problem 5.9** **(solution page 612)**

The traditional implementation of the merge step of mergesort requires three loops [98]:

```
1   void merge(long src1[], long src2[], long dest[], long n) {
2       long i1 = 0;
3       long i2 = 0;
4       long id = 0;
5       while (i1 < n && i2 < n) {
6           if (src1[i1] < src2[i2])
7               dest[id++] = src1[i1++];
8           else
9               dest[id++] = src2[i2++];
10      }
11      while (i1 < n)
12          dest[id++] = src1[i1++];
13      while (i2 < n)
14          dest[id++] = src2[i2++];
15  }
```

The branches caused by comparing variables `i1` and `i2` to `n` have good prediction performance—the only mispredictions occur when they first become false. The comparison between values `src1[i1]` and `src2[i2]` (line 6), on the other hand, is highly unpredictable for typical data. This comparison controls a conditional branch, yielding a CPE (where the number of elements is $2n$) of around 15.0 when run on random data.

Rewrite the code so that the effect of the conditional statement in the first loop (lines 6–9) can be implemented with a conditional move.

## 5.12 Understanding Memory Performance

All of the code we have written thus far, and all the tests we have run, access relatively small amounts of memory. For example, the combining routines were measured over vectors of length less than 1,000 elements, requiring no more than 8,000 bytes of data. All modern processors contain one or more *cache* memories to provide fast access to such small amounts of memory. In this section, we will further investigate the performance of programs that involve load (reading from memory into registers) and store (writing from registers to memory) operations, considering only the cases where all data are held in cache. In Chapter 6, we go into much more detail about how caches work, their performance characteristics, and how to write code that makes best use of caches.

As Figure 5.11 shows, modern processors have dedicated functional units to perform load and store operations, and these units have internal buffers to hold sets of outstanding requests for memory operations. For example, our reference machine has two load units, each of which can hold up to 72 pending read requests. It has a single store unit with a store buffer containing up to 42 write requests. Each of these units can initiate 1 operation every clock cycle.

### 5.12.1 Load Performance

The performance of a program containing load operations depends on both the pipelining capability and the latency of the load unit. In our experiments with combining operations using our reference machine, we saw that the CPE never got below 0.50 for any combination of data type and combining operation, except when using SIMD operations. One factor limiting the CPE for our examples is that they all require reading one value from memory for each element computed. With two load units, each able to initiate at most 1 load operation every clock cycle, the CPE cannot be less than 0.50. For applications where we must load $k$ values for every element computed, we can never achieve a CPE lower than $k/2$ (see, for example, Problem 5.15).

In our examples so far, we have not seen any performance effects due to the latency of load operations. The addresses for our load operations depended only on the loop index $i$, and so the load operations did not form part of a performance-limiting critical path.

To determine the latency of the load operation on a machine, we can set up a computation with a sequence of load operations, where the outcome of one determines the address for the next. As an example, consider the function `list_len` in Figure 5.31, which computes the length of a linked list. In the loop of this function, each successive value of variable `ls` depends on the value read by the pointer reference `ls->next`. Our measurements show that function `list_len` has

```
1   typedef struct ELE {
2       struct ELE *next;
3       long data;
4   } list_ele, *list_ptr;
5
6   long list_len(list_ptr ls) {
7       long len = 0;
8       while (ls) {
9         len++;
10        ls = ls->next;
11      }
12      return len;
13  }
```

**Figure 5.31 Linked list function.** Its performance is limited by the latency of the load operation.

a CPE of 4.00, which we claim is a direct indication of the latency of the load operation. To see this, consider the assembly code for the loop:

```
    Inner loop of list_len
    ls in %rdi, len in %rax
1   .L3:                        loop:
2     addq    $1, %rax            Increment len
3     movq    (%rdi), %rdi        ls = ls->next
4     testq   %rdi, %rdi          Test ls
5     jne     .L3                 If nonnull, goto loop
```

The `movq` instruction on line 3 forms the critical bottleneck in this loop. Each successive value of register `%rdi` depends on the result of a load operation having the value in `%rdi` as its address. Thus, the load operation for one iteration cannot begin until the one for the previous iteration has completed. The CPE of 4.00 for this function is determined by the latency of the load operation. Indeed, this measurement matches the documented access time of 4 cycles for the reference machine's L1 cache, as is discussed in Section 6.4.

### 5.12.2 Store Performance

In all of our examples thus far, we analyzed only functions that reference memory mostly with load operations, reading from a memory location into a register. Its counterpart, the *store* operation, writes a register value to memory. The performance of this operation, particularly in relation to its interactions with load operations, involves several subtle issues.

As with the load operation, in most cases, the store operation can operate in a fully pipelined mode, beginning a new store on every cycle. For example, consider the function shown in Figure 5.32 that sets the elements of an array `dest` of length `n` to zero. Our measurements show a CPE of 1.0. This is the best we can achieve on a machine with a single store functional unit.

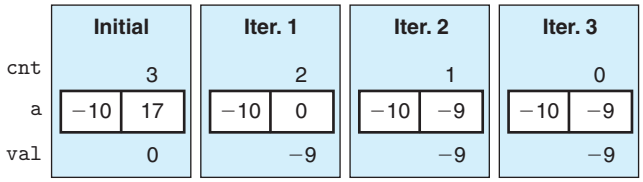
Unlike the other operations we have considered so far, the store operation does not affect any register values. Thus, by their very nature, a series of store operations cannot create a data dependency. Only a load operation is affected by the result of a store operation, since only a load can read back the memory value that has been written by the store. The function `write_read` shown in Figure 5.33

```
1  /* Set elements of array to 0 */
2  void clear_array(long *dest, long n) {
3      long i;
4      for (i = 0; i < n; i++)
5          dest[i] = 0;
6  }
```

**Figure 5.32 Function to set array elements to 0.** This code achieves a CPE of 1.0.

```
1   /* Write to dest, read from src */
2   void write_read(long *src, long *dst, long n)
3   {
4       long cnt = n;
5       long val = 0;
6
7       while (cnt) {
8           *dst = val;
9           val = (*src)+1;
10          cnt--;
11      }
12  }
```

**Example A:** `write_read(&a[0],&a[1],3)`




**Example B:** `write_read(&a[0],&a[0],3)`

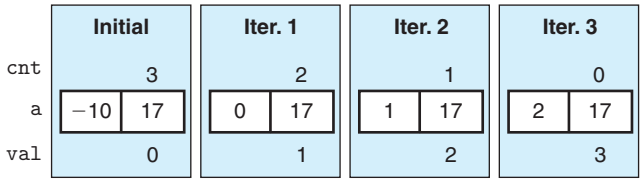



**Figure 5.33 Code to write and read memory locations, along with illustrative executions.** This function highlights the interactions between stores and loads when arguments `src` and `dest` are equal.

illustrates the potential interactions between loads and stores. This figure also shows two example executions of this function, when it is called for a two-element array `a`, with initial contents −10 and 17, and with argument `cnt` equal to 3. These executions illustrate some subtleties of the load and store operations.

In Example A of Figure 5.33, argument `src` is a pointer to array element `a[0]`, while `dest` is a pointer to array element `a[1]`. In this case, each load by the pointer reference `*src` will yield the value −10. Hence, after two iterations, the array elements will remain fixed at −10 and −9, respectively. The result of the read from `src` is not affected by the write to `dest`. Measuring this example over a larger number of iterations gives a CPE of 1.3.

In Example B of Figure 5.33, both arguments `src` and `dest` are pointers to array element `a[0]`. In this case, each load by the pointer reference `*src` will yield the value stored by the previous execution of the pointer reference `*dest`.

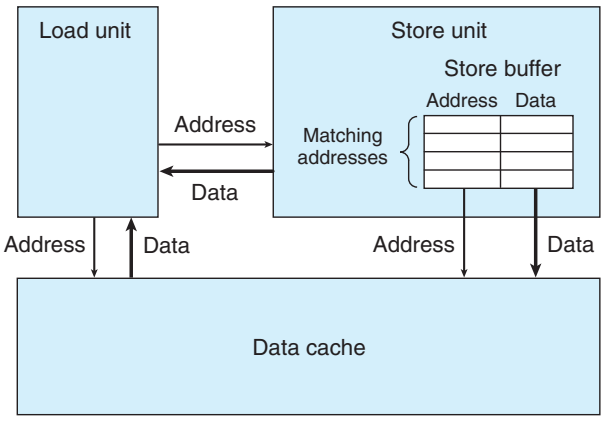



**Figure 5.34**
**Detail of load and store units.** The store unit maintains a buffer of pending writes. The load unit must check its address with those in the store unit to detect a write/read dependency.

As a consequence, a series of ascending values will be stored in this location. In general, if function `write_read` is called with arguments `src` and `dest` pointing to the same memory location, and with argument `cnt` having some value $n > 0$, the net effect is to set the location to $n - 1$. This example illustrates a phenomenon we will call a *write/read dependency*—the outcome of a memory read depends on a recent memory write. Our performance measurements show that Example B has a CPE of 7.3. The write/read dependency causes a slowdown in the processing of around 6 clock cycles.

To see how the processor can distinguish between these two cases and why one runs slower than the other, we must take a more detailed look at the load and store execution units, as shown in Figure 5.34. The store unit includes a *store buffer* containing the addresses and data of the store operations that have been issued to the store unit, but have not yet been completed, where completion involves updating the data cache. This buffer is provided so that a series of store operations can be executed without having to wait for each one to update the cache. When a load operation occurs, it must check the entries in the store buffer for matching addresses. If it finds a match (meaning that any of the bytes being written have the same address as any of the bytes being read), it retrieves the corresponding data entry as the result of the load operation.

GCC generates the following code for the inner loop of `write_read`:

```
  Inner loop of write_read
  src in %rdi, dst in %rsi, val in %rax
.L3:                          loop:
  movq    %rax, (%rsi)          Write val to dst
  movq    (%rdi), %rax          t = *src
  addq    $1, %rax              val = t+1
  subq    $1, %rdx              cnt--
  jne     .L3                   If != 0, goto loop
```

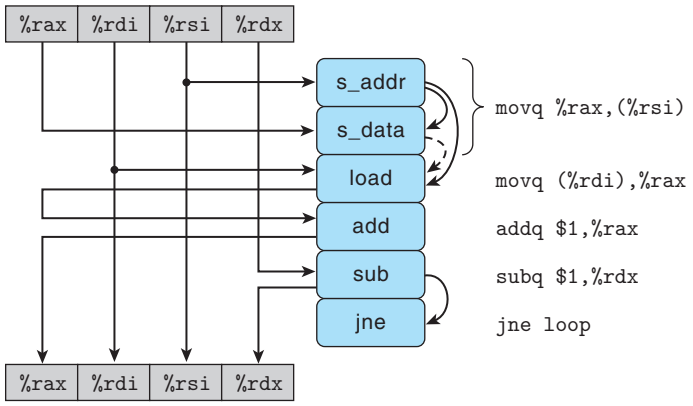



**Figure 5.35**
**Graphical representation of inner-loop code for `write_read`.** The first `movl` instruction is decoded into separate operations to compute the store address and to store the data to memory.

Figure 5.35 shows a data-flow representation of this loop code. The instruction `movq %rax,(%rsi)` is translated into two operations: The s_addr instruction computes the address for the store operation, creates an entry in the store buffer, and sets the address field for that entry. The s_data operation sets the data field for the entry. As we will see, the fact that these two computations are performed independently can be important to program performance. This motivates the separate functional units for these operations in the reference machine.

In addition to the data dependencies between the operations caused by the writing and reading of registers, the arcs on the right of the operators denote a set of implicit dependencies for these operations. In particular, the address computation of the s_addr operation must clearly precede the s_data operation. In addition, the load operation generated by decoding the instruction `movq (%rdi), %rax` must check the addresses of any pending store operations, creating a data dependency between it and the s_addr operation. The figure shows a dashed arc between the s_data and load operations. This dependency is conditional: if the two addresses match, the load operation must wait until the s_data has deposited its result into the store buffer, but if the two addresses differ, the two operations can proceed independently.

Figure 5.36 illustrates the data dependencies between the operations for the inner loop of `write_read`. In Figure 5.36(a), we have rearranged the operations to allow the dependencies to be seen more clearly. We have labeled the three dependencies involving the load and store operations for special attention. The arc labeled "1" represents the requirement that the store address must be computed before the data can be stored. The arc labeled "2" represents the need for the load operation to compare its address with that for any pending store operations. Finally, the dashed arc labeled "3" represents the conditional data dependency that arises when the load and store addresses match.

Figure 5.36(b) illustrates what happens when we take away those operations that do not directly affect the flow of data from one iteration to the next. The data-flow graph shows just two chains of dependencies: the one on the left, with data values being stored, loaded, and incremented (only for the case of matching addresses); and the one on the right, decrementing variable `cnt`.

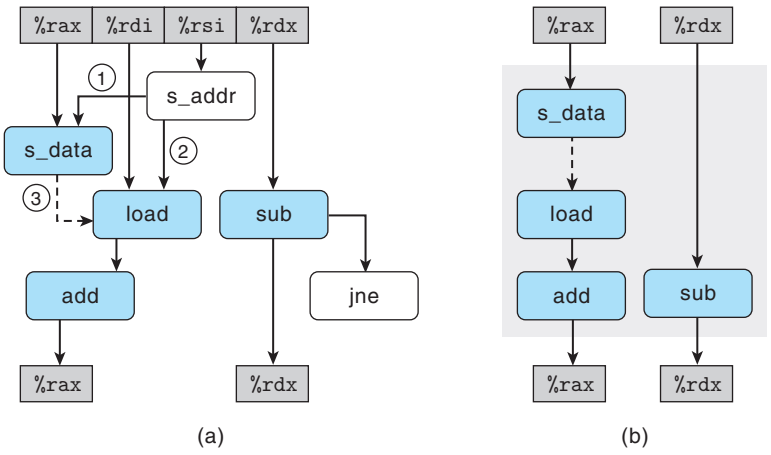



**Figure 5.36**
**Abstracting the operations for `write_read`.** We first rearrange the operators of Figure 5.35(a) and then show only those operations that use values from one iteration to produce new values for the next (b).

We can now understand the performance characteristics of function `write_read`. Figure 5.37 illustrates the data dependencies formed by multiple iterations of its inner loop. For the case of Example A in Figure 5.33, with differing source and destination addresses, the load and store operations can proceed independently, and hence the only critical path is formed by the decrementing of variable `cnt`, resulting in a CPE bound of 1.0. For the case of Example B with matching source and destination addresses, the data dependency between the s_data and load instructions causes a critical path to form involving data being stored, loaded, and incremented. We found that these three operations in sequence require a total of around 7 clock cycles.

As these two examples show, the implementation of memory operations involves many subtleties. With operations on registers, the processor can determine which instructions will affect which others as they are being decoded into operations. With memory operations, on the other hand, the processor cannot predict which will affect which others until the load and store addresses have been computed. Efficient handling of memory operations is critical to the performance of many programs. The memory subsystem makes use of many optimizations, such as the potential parallelism when operations can proceed independently.

### Practice Problem 5.10 (solution page 613)

As another example of code with potential load-store interactions, consider the following function to copy the contents of one array to another:

```
1  void copy_array(long *src, long *dest, long n)
2  {
3      long i;
4      for (i = 0; i < n; i++)
5          dest[i] = src[i];
6  }
```

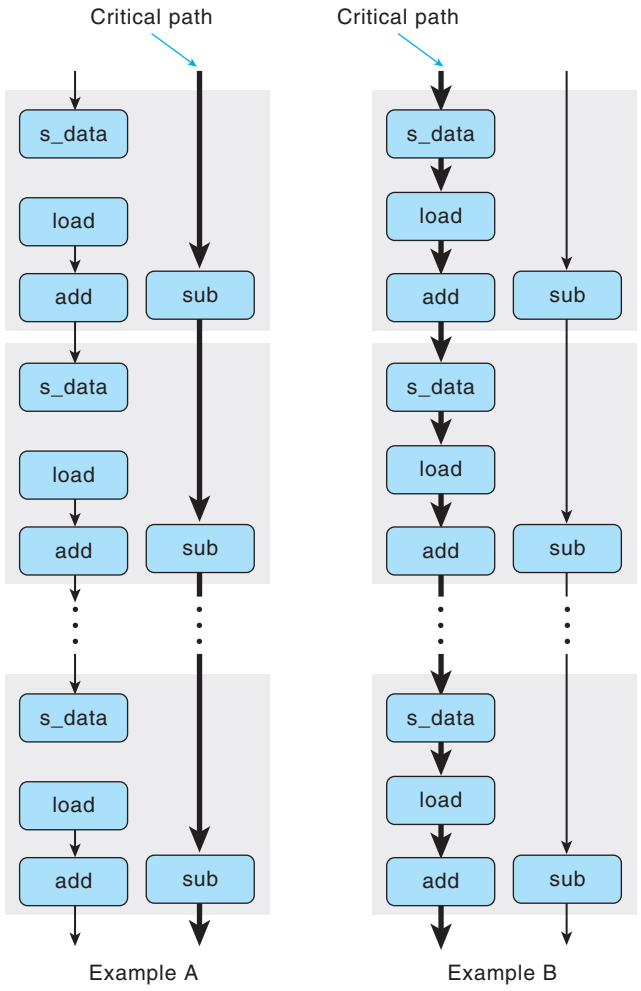



**Figure 5.37**
**Data-flow representation of function `write_read`.** When the two addresses do not match, the only critical path is formed by the decrementing of `cnt` (Example A). When they do match, the chain of data being stored, loaded, and incremented forms the critical path (Example B).

Suppose `a` is an array of length 1,000 initialized so that each element `a[`$i$`]` equals $i$.

A. What would be the effect of the call `copy_array(a+1,a,999)`?

B. What would be the effect of the call `copy_array(a,a+1,999)`?

C. Our performance measurements indicate that the call of part A has a CPE of 1.2 (which drops to 1.0 when the loop is unrolled by a factor of 4), while the call of part B has a CPE of 5.0. To what factor do you attribute this performance difference?

D. What performance would you expect for the call `copy_array(a,a,999)`?

**Practice Problem 5.11** (solution page 613)

We saw that our measurements of the prefix-sum function `psum1` (Figure 5.1) yield a CPE of 9.00 on a machine where the basic operation to be performed, floating-point addition, has a latency of just 3 clock cycles. Let us try to understand why our function performs so poorly.

The following is the assembly code for the inner loop of the function:

```
    Inner loop of psum1
    a in %rdi, i in %rax, cnt in %rdx
1   .L5:                                          loop:
2     vmovss  -4(%rsi,%rax,4), %xmm0                Get p[i-1]
3     vaddss  (%rdi,%rax,4), %xmm0, %xmm0           Add a[i]
4     vmovss  %xmm0, (%rsi,%rax,4)                  Store at p[i]
5     addq    $1, %rax                              Increment i
6     cmpq    %rdx, %rax                            Compare i:cnt
7     jne     .L5                                   If !=, goto loop
```

Perform an analysis similar to those shown for `combine3` (Figure 5.14) and for `write_read` (Figure 5.36) to diagram the data dependencies created by this loop, and hence the critical path that forms as the computation proceeds. Explain why the CPE is so high.

**Practice Problem 5.12** (solution page 613)

Rewrite the code for `psum1` (Figure 5.1) so that it does not need to repeatedly retrieve the value of `p[i]` from memory. You do not need to use loop unrolling. We measured the resulting code to have a CPE of 3.00, limited by the latency of floating-point addition.

## 5.13 Life in the Real World: Performance Improvement Techniques

Although we have only considered a limited set of applications, we can draw important lessons on how to write efficient code. We have described a number of basic strategies for optimizing program performance:

*High-level design.* Choose appropriate algorithms and data structures for the problem at hand. Be especially vigilant to avoid algorithms or coding techniques that yield asymptotically poor performance.

*Basic coding principles.* Avoid optimization blockers so that a compiler can generate efficient code.

- Eliminate excessive function calls. Move computations out of loops when possible. Consider selective compromises of program modularity to gain greater efficiency.

- Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.

*Low-level optimizations.* Structure code to take advantage of the hardware capabilities.

- Unroll loops to reduce overhead and to enable further optimizations.
- Find ways to increase instruction-level parallelism by techniques such as multiple accumulators and reassociation.
- Rewrite conditional operations in a functional style to enable compilation via conditional data transfers.

A final word of advice to the reader is to be vigilant to avoid introducing errors as you rewrite programs in the interest of efficiency. It is very easy to make mistakes when introducing new variables, changing loop bounds, and making the code more complex overall. One useful technique is to use checking code to test each version of a function as it is being optimized, to ensure no bugs are introduced during this process. Checking code applies a series of tests to the new versions of a function and makes sure they yield the same results as the original. The set of test cases must become more extensive with highly optimized code, since there are more cases to consider. For example, checking code that uses loop unrolling requires testing for many different loop bounds to make sure it handles all of the different possible numbers of single-step iterations required at the end.

## 5.14 Identifying and Eliminating Performance Bottlenecks

Up to this point, we have only considered optimizing small programs, where there is some clear place in the program that limits its performance and therefore should be the focus of our optimization efforts. When working with large programs, even knowing where to focus our optimization efforts can be difficult. In this section, we describe how to use *code profilers*, analysis tools that collect performance data about a program as it executes. We also discuss some general principles of code optimization, including the implications of Amdahl's law, introduced in Section 1.9.1.

### 5.14.1 Program Profiling

Program *profiling* involves running a version of a program in which instrumentation code has been incorporated to determine how much time the different parts of the program require. It can be very useful for identifying the parts of a program we should focus on in our optimization efforts. One strength of profiling is that it can be performed while running the actual program on realistic benchmark data.

Unix systems provide the profiling program GPROF. This program generates two forms of information. First, it determines how much CPU time was spent for each of the functions in the program. Second, it computes a count of how many times each function gets called, categorized by which function performs the call. Both forms of information can be quite useful. The timings give a sense of

the relative importance of the different functions in determining the overall run time. The calling information allows us to understand the dynamic behavior of the program.

Profiling with GPROF requires three steps, as shown for a C program `prog.c`, which runs with command-line argument `file.txt`:

1. The program must be compiled and linked for profiling. With GCC (and other C compilers), this involves simply including the run-time flag `-pg` on the command line. It is important to ensure that the compiler does not attempt to perform any optimizations via inline substitution, or else the calls to functions may not be tabulated accurately. We use optimization flag `-Og`, guaranteeing that function calls will be tracked properly.

   ```
   linux> gcc -Og -pg prog.c -o prog
   ```

2. The program is then executed as usual:

   ```
   linux> ./prog file.txt
   ```

   It runs slightly (around a factor of 2) slower than normal, but otherwise the only difference is that it generates a file `gmon.out`.

3. GPROF is invoked to analyze the data in `gmon.out`:

   ```
   linux> gprof prog
   ```

The first part of the profile report lists the times spent executing the different functions, sorted in descending order. As an example, the following listing shows this part of the report for the three most time-consuming functions in a program:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 97.58    203.66   203.66        1   203.66   203.66  sort_words
  2.32    208.50     4.85   965027     0.00     0.00  find_ele_rec
  0.14    208.81     0.30 12511031     0.00     0.00  Strlen
```

Each row represents the time spent for all calls to some function. The first column indicates the percentage of the overall time spent on the function. The second shows the cumulative time spent by the functions up to and including the one on this row. The third shows the time spent on this particular function, and the fourth shows how many times it was called (not counting recursive calls). In our example, the function `sort_words` was called only once, but this single call required 203.66 seconds, while the function `find_ele_rec` was called 965,027 times (not including recursive calls), requiring a total of 4.85 seconds. Function `Strlen` computes the length of a string by calling the library function `strlen`. Library function calls are normally not shown in the results by GPROF. Their times are usually reported as part of the function calling them. By creating the "wrapper function" `Strlen`, we can reliably track the calls to `strlen`, showing that it was called 12,511,031 times but only requiring a total of 0.30 seconds.

The second part of the profile report shows the calling history of the functions. The following is the history for a recursive function `find_ele_rec`:

```
                                158655725             find_ele_rec [5]
                4.85    0.10  965027/965027      insert_string [4]
[5]      2.4    4.85    0.10  965027+158655725 find_ele_rec [5]
                0.08    0.01  363039/363039      save_string [8]
                0.00    0.01  363039/363039      new_ele [12]
                                158655725             find_ele_rec [5]
```

This history shows both the functions that called `find_ele_rec`, as well as the functions that it called. The first two lines show the calls to the function: 158,655,725 calls by itself recursively, and 965,027 calls by function `insert_string` (which is itself called 965,027 times). Function `find_ele_rec`, in turn, called two other functions, `save_string` and `new_ele`, each a total of 363,039 times.

From these call data, we can often infer useful information about the program behavior. For example, the function `find_ele_rec` is a recursive procedure that scans the linked list for a hash bucket looking for a particular string. For this function, comparing the number of recursive calls with the number of top-level calls provides statistical information about the lengths of the traversals through these lists. Given that their ratio is 164.4:1, we can infer that the program scanned an average of around 164 elements each time.

Some properties of GPROF are worth noting:

- The timing is not very precise. It is based on a simple *interval counting* scheme in which the compiled program maintains a counter for each function recording the time spent executing that function. The operating system causes the program to be interrupted at some regular time interval $\delta$. Typical values of $\delta$ range between 1.0 and 10.0 milliseconds. It then determines what function the program was executing when the interrupt occurred and increments the counter for that function by $\delta$. Of course, it may happen that this function just started executing and will shortly be completed, but it is assigned the full cost of the execution since the previous interrupt. Some other function may run between two interrupts and therefore not be charged any time at all.

  Over a long duration, this scheme works reasonably well. Statistically, every function should be charged according to the relative time spent executing it. For programs that run for less than around 1 second, however, the numbers should be viewed as only rough estimates.
- The calling information is quite reliable, assuming no inline substitutions have been performed. The compiled program maintains a counter for each combination of caller and callee. The appropriate counter is incremented every time a procedure is called.
- By default, the timings for library functions are not shown. Instead, these times are incorporated into the times for the calling functions.

### 5.14.2 Using a Profiler to Guide Optimization

As an example of using a profiler to guide program optimization, we created an application that involves several different tasks and data structures. This application analyzes the *n-gram* statistics of a text document, where an $n$-gram is a sequence of $n$ words occurring in a document. For $n = 1$, we collect statistics on individual words, for $n = 2$ on pairs of words, and so on. For a given value of $n$, our program reads a text file, creates a table of unique $n$-grams and how many times each one occurs, then sorts the $n$-grams in descending order of occurrence.

As a benchmark, we ran it on a file consisting of the complete works of William Shakespeare, totaling 965,028 words, of which 23,706 are unique. We found that for $n = 1$, even a poorly written analysis program can readily process the entire file in under 1 second, and so we set $n = 2$ to make things more challenging. For the case of $n = 2$, $n$-grams are referred to as *bigrams* (pronounced "bye-grams"). We determined that Shakespeare's works contain 363,039 unique bigrams. The most common is "I am," occurring 1,892 times. Perhaps his most famous bigram, "to be," occurs 1,020 times. Fully 266,018 of the bigrams occur only once.

Our program consists of the following parts. We created multiple versions, starting with simple algorithms for the different parts and then replacing them with more sophisticated ones:

1. Each word is read from the file and converted to lowercase. Our initial version used the function `lower1` (Figure 5.7), which we know to have quadratic run time due to repeated calls to `strlen`.
2. A hash function is applied to the string to create a number between 0 and $s - 1$, for a hash table with $s$ buckets. Our initial function simply summed the ASCII codes for the characters modulo $s$.
3. Each hash bucket is organized as a linked list. The program scans down this list looking for a matching entry. If one is found, the frequency for this $n$-gram is incremented. Otherwise, a new list element is created. Our initial version performed this operation recursively, inserting new elements at the end of the list.
4. Once the table has been generated, we sort all of the elements according to the frequencies. Our initial version used insertion sort.

Figure 5.38 shows the profile results for six different versions of our $n$-gram-frequency analysis program. For each version, we divide the time into the following categories:

Sort. Sorting $n$-grams by frequency

List. Scanning the linked list for a matching $n$-gram, inserting a new element if necessary

Lower. Converting strings to lowercase

Strlen. Computing string lengths

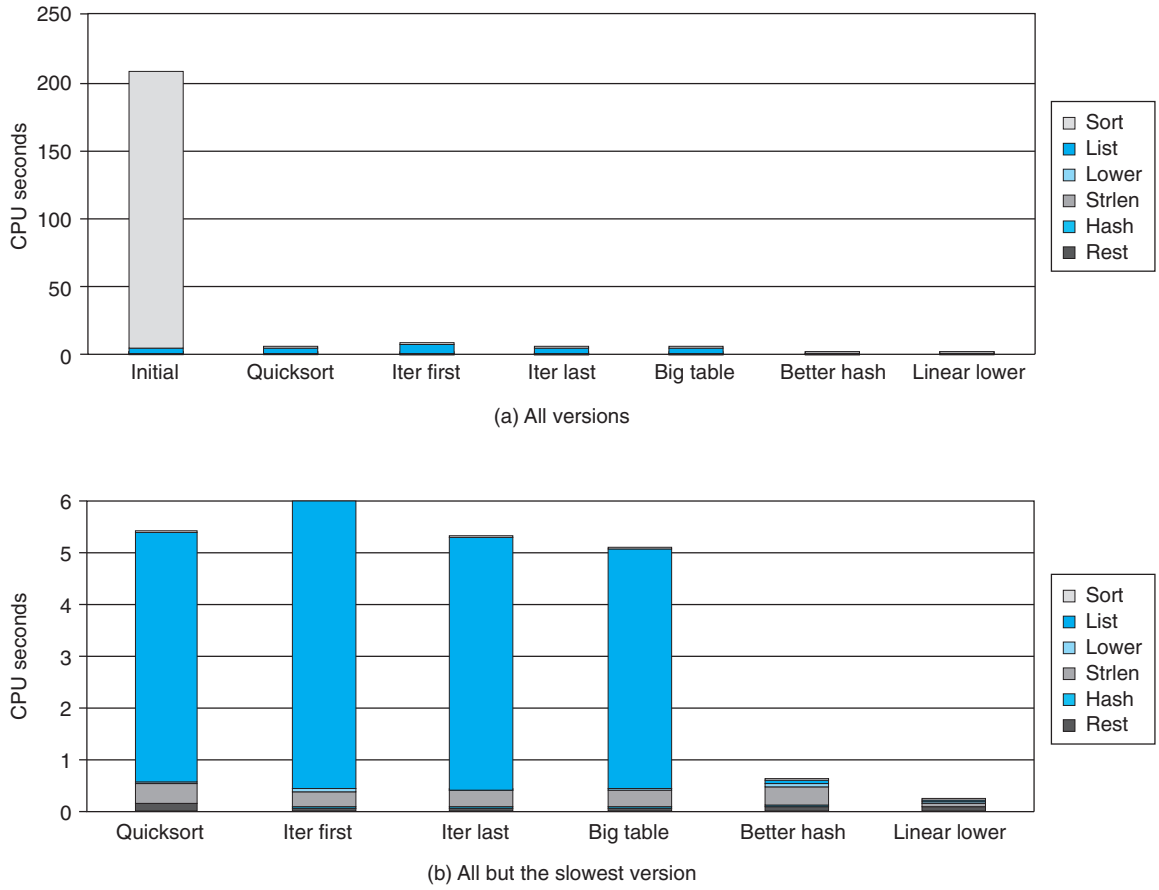



**Figure 5.38 Profile results for different versions of bigram-frequency counting program.** Time is divided according to the different major operations in the program.

Hash. Computing the hash function

Rest. The sum of all other functions

As part (a) of the figure shows, our initial version required 3.5 minutes, with most of the time spent sorting. This is not surprising, since insertion sort has quadratic run time and the program sorted 363,039 values.

In our next version, we performed sorting using the library function `qsort`, which is based on the quicksort algorithm [98]. It has an expected run time of $O(n \log n)$. This version is labeled "Quicksort" in the figure. The more efficient sorting algorithm reduces the time spent sorting to become negligible, and the overall run time to around 5.4 seconds. Part (b) of the figure shows the times for the remaining version on a scale where we can see them more clearly.

With improved sorting, we now find that list scanning becomes the bottleneck. Thinking that the inefficiency is due to the recursive structure of the function, we replaced it by an iterative one, shown as "Iter first." Surprisingly, the run time increases to around 7.5 seconds. On closer study, we find a subtle difference between the two list functions. The recursive version inserted new elements at the end of the list, while the iterative one inserted them at the front. To maximize performance, we want the most frequent $n$-grams to occur near the beginning of the lists. That way, the function will quickly locate the common cases. Assuming that $n$-grams are spread uniformly throughout the document, we would expect the first occurrence of a frequent one to come before that of a less frequent one. By inserting new $n$-grams at the end, the first function tended to order $n$-grams in descending order of frequency, while the second function tended to do just the opposite. We therefore created a third list-scanning function that uses iteration but inserts new elements at the end of this list. With this version, shown as "Iter last," the time dropped to around 5.3 seconds, slightly better than with the recursive version. These measurements demonstrate the importance of running experiments on a program as part of an optimization effort. We initially assumed that converting recursive code to iterative code would improve its performance and did not consider the distinction between adding to the end or to the beginning of a list.

Next, we consider the hash table structure. The initial version had only 1,021 buckets (typically, the number of buckets is chosen to be a prime number to enhance the ability of the hash function to distribute keys uniformly among the buckets). For a table with 363,039 entries, this would imply an average *load* of $363{,}039/1{,}021 = 355.6$. That explains why so much of the time is spent performing list operations—the searches involve testing a significant number of candidate $n$-grams. It also explains why the performance is so sensitive to the list ordering. We then increased the number of buckets to 199,999, reducing the average load to 1.8. Oddly enough, however, our overall run time only drops to 5.1 seconds, a difference of only 0.2 seconds.

On further inspection, we can see that the minimal performance gain with a larger table was due to a poor choice of hash function. Simply summing the character codes for a string does not produce a very wide range of values. In particular, the maximum code value for a letter is 122, and so a string of $n$ characters will generate a sum of at most $122n$. The longest bigram in our document, "honorificabilitudinitatibus thou" sums to just 3,371, and so most of the buckets in our hash table will go unused. In addition, a commutative hash function, such as addition, does not differentiate among the different possible orderings of characters with a string. For example, the words "rat" and "tar" will generate the same sums.

We switched to a hash function that uses shift and EXCLUSIVE-OR operations. With this version, shown as "Better hash," the time drops to 0.6 seconds. A more systematic approach would be to study the distribution of keys among the buckets more carefully, making sure that it comes close to what one would expect if the hash function had a uniform output distribution.

Finally, we have reduced the run time to the point where most of the time is spent in `strlen`, and most of the calls to `strlen` occur as part of the lowercase conversion. We have already seen that function `lower1` has quadratic performance, especially for long strings. The words in this document are short enough to avoid the disastrous consequences of quadratic performance; the longest bigram is just 32 characters. Still, switching to `lower2`, shown as "Linear lower," yields a significant improvement, with the overall time dropping to around 0.2 seconds.

With this exercise, we have shown that code profiling can help drop the time required for a simple application from 3.5 minutes down to 0.2 seconds, yielding a performance gain of around 1,000×. The profiler helps us focus our attention on the most time-consuming parts of the program and also provides useful information about the procedure call structure. Some of the bottlenecks in our code, such as using a quadratic sort routine, are easy to anticipate, while others, such as whether to append to the beginning or end of a list, emerge only through a careful analysis.

We can see that profiling is a useful tool to have in the toolbox, but it should not be the only one. The timing measurements are imperfect, especially for shorter (less than 1 second) run times. More significantly, the results apply only to the particular data tested. For example, if we had run the original function on data consisting of a smaller number of longer strings, we would have found that the lowercase conversion routine was the major performance bottleneck. Even worse, if it only profiled documents with short words, we might never detect hidden bottlenecks such as the quadratic performance of `lower1`. In general, profiling can help us optimize for *typical* cases, assuming we run the program on representative data, but we should also make sure the program will have respectable performance for all possible cases. This mainly involves avoiding algorithms (such as insertion sort) and bad programming practices (such as `lower1`) that yield poor asymptotic performance.

Amdahl's law, described in Section 1.9.1, provides some additional insights into the performance gains that can be obtained by targeted optimizations. For our $n$-gram code, we saw the total execution time drop from 209.0 to 5.4 seconds when we replaced insertion sort by quicksort. The initial version spent 203.7 of its 209.0 seconds performing insertion sort, giving $\alpha = 0.974$, the fraction of time subject to speedup. With quicksort, the time spent sorting becomes negligible, giving a predicted speedup of $209/\alpha = 39.0$, close to the measured speedup of 38.5. We were able to gain a large speedup because sorting constituted a very large fraction of the overall execution time. However, when one bottleneck is eliminated, a new one arises, and so gaining additional speedup required focusing on other parts of the program.

## 5.15 Summary

Although most presentations on code optimization describe how compilers can generate efficient code, much can be done by an application programmer to assist the compiler in this task. No compiler can replace an inefficient algorithm or data

structure by a good one, and so these aspects of program design should remain a primary concern for programmers. We also have seen that optimization blockers, such as memory aliasing and procedure calls, seriously restrict the ability of compilers to perform extensive optimizations. Again, the programmer must take primary responsibility for eliminating these. These should simply be considered parts of good programming practice, since they serve to eliminate unneeded work.

Tuning performance beyond a basic level requires some understanding of the processor's microarchitecture, describing the underlying mechanisms by which the processor implements its instruction set architecture. For the case of out-of-order processors, just knowing something about the operations, capabilities, latencies, and issue times of the functional units establishes a baseline for predicting program performance.

We have studied a series of techniques—including loop unrolling, creating multiple accumulators, and reassociation—that can exploit the instruction-level parallelism provided by modern processors. As we get deeper into the optimization, it becomes important to study the generated assembly code and to try to understand how the computation is being performed by the machine. Much can be gained by identifying the critical paths determined by the data dependencies in the program, especially between the different iterations of a loop. We can also compute a throughput bound for a computation, based on the number of operations that must be computed and the number and issue times of the units that perform those operations.

Programs that involve conditional branches or complex interactions with the memory system are more difficult to analyze and optimize than the simple loop programs we first considered. The basic strategy is to try to make branches more predictable or make them amenable to implementation using conditional data transfers. We must also watch out for the interactions between store and load operations. Keeping values in local variables, allowing them to be stored in registers, can often be helpful.

When working with large programs, it becomes important to focus our optimization efforts on the parts that consume the most time. Code profilers and related tools can help us systematically evaluate and improve program performance. We described GPROF, a standard Unix profiling tool. More sophisticated profilers are available, such as the VTUNE program development system from Intel, and VALGRIND, commonly available on Linux systems. These tools can break down the execution time below the procedure level to estimate the performance of each *basic block* of the program. (A basic block is a sequence of instructions that has no transfers of control out of its middle, and so the block is always executed in its entirety.)

## Bibliographic Notes

Our focus has been to describe code optimization from the programmer's perspective, demonstrating how to write code that will make it easier for compilers to generate efficient code. An extended paper by Chellappa, Franchetti, and Püschel [19]

takes a similar approach but goes into more detail with respect to the processor's characteristics.

Many publications describe code optimization from a compiler's perspective, formulating ways that compilers can generate more efficient code. Muchnick's book is considered the most comprehensive [80]. Wadleigh and Crawford's book on software optimization [115] covers some of the material we have presented, but it also describes the process of getting high performance on parallel machines. An early paper by Mahlke et al. [75] describes how several techniques developed for compilers that map programs onto parallel machines can be adapted to exploit the instruction-level parallelism of modern processors. This paper covers the code transformations we presented, including loop unrolling, multiple accumulators (which they refer to as *accumulator variable expansion*), and reassociation (which they refer to as *tree height reduction*).

Our presentation of the operation of an out-of-order processor is fairly brief and abstract. More complete descriptions of the general principles can be found in advanced computer architecture textbooks, such as the one by Hennessy and Patterson [46, Ch. 2–3]. Shen and Lipasti's book [100] provides an in-depth treatment of modern processor design.

## Homework Problems

### 5.13 ◆◆

Suppose we wish to write a procedure that computes the inner product of two vectors `u` and `v`. An abstract version of the function has a CPE of 14–18 with x86-64 for different types of integer and floating-point data. By doing the same sort of transformations we did to transform the abstract program `combine1` into the more efficient `combine4`, we get the following code:

```
1   /* Inner product.  Accumulate in temporary */
2   void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(u);
6       data_t *udata = get_vec_start(u);
7       data_t *vdata = get_vec_start(v);
8       data_t sum = (data_t) 0;
9
10      for (i = 0; i < length; i++) {
11          sum = sum + udata[i] * vdata[i];
12      }
13      *dest = sum;
14  }
```

Our measurements show that this function has CPEs of 1.50 for integer data and 3.00 for floating-point data. For data type `double`, the x86-64 assembly code for the inner loop is as follows:

```
    Inner loop of inner4.  data_t = double, OP = *
    udata in %rbp, vdata in %rax, sum in %xmm0
    i in %rcx, limit in %rbx
1   .L15:                                       loop:
2     vmovsd  0(%rbp,%rcx,8), %xmm1               Get udata[i]
3     vmulsd  (%rax,%rcx,8), %xmm1, %xmm1         Multiply by vdata[i]
4     vaddsd  %xmm1, %xmm0, %xmm0                 Add to sum
5     addq    $1, %rcx                            Increment i
6     cmpq    %rbx, %rcx                          Compare i:limit
7     jne     .L15                                If !=, goto loop
```

Assume that the functional units have the characteristics listed in Figure 5.12.

A. Diagram how this instruction sequence would be decoded into operations and show how the data dependencies between them would create a critical path of operations, in the style of Figures 5.13 and 5.14.

B. For data type `double`, what lower bound on the CPE is determined by the critical path?

C. Assuming similar instruction sequences for the integer code as well, what lower bound on the CPE is determined by the critical path for integer data?

D. Explain how the floating-point versions can have CPEs of 3.00, even though the multiplication operation requires 5 clock cycles.

**5.14** ◆

Write a version of the inner product procedure described in Problem 5.13 that uses $6 \times 1$ loop unrolling. For x86-64, our measurements of the unrolled version give a CPE of 1.07 for integer data but still 3.01 for both floating-point data.

A. Explain why any (scalar) version of an inner product procedure running on an Intel Core i7 Haswell processor cannot achieve a CPE less than 1.00.

B. Explain why the performance for floating-point data did not improve with loop unrolling.

**5.15** ◆

Write a version of the inner product procedure described in Problem 5.13 that uses $6 \times 6$ loop unrolling. Our measurements for this function with x86-64 give a CPE of 1.06 for integer data and 1.01 for floating-point data.

What factor limits the performance to a CPE of 1.00?

**5.16** ◆

Write a version of the inner product procedure described in Problem 5.13 that uses $6 \times 1a$ loop unrolling to enable greater parallelism. Our measurements for this function give a CPE of 1.10 for integer data and 1.05 for floating-point data.

**5.17** ◆◆

The library function `memset` has the following prototype:

```
void *memset(void *s, int c, size_t n);
```

This function fills `n` bytes of the memory area starting at `s` with copies of the low-order byte of `c`. For example, it can be used to zero out a region of memory by giving argument 0 for `c`, but other values are possible.

The following is a straightforward implementation of `memset`:

```
1  /* Basic implementation of memset */
2  void *basic_memset(void *s, int c, size_t n)
3  {
4      size_t cnt = 0;
5      unsigned char *schar = s;
6      while (cnt < n) {
7          *schar++ = (unsigned char) c;
8          cnt++;
9      }
10     return s;
11 }
```

Implement a more efficient version of the function by using a word of data type `unsigned long` to pack eight copies of `c`, and then step through the region using word-level writes. You might find it helpful to do additional loop unrolling as well. On our reference machine, we were able to reduce the CPE from 1.00 for the straightforward implementation to 0.127. That is, the program is able to write 8 bytes every clock cycle.

Here are some additional guidelines. To ensure portability, let $K$ denote the value of `sizeof(unsigned long)` for the machine on which you run your program.

- You may not call any library functions.
- Your code should work for arbitrary values of `n`, including when it is not a multiple of $K$. You can do this in a manner similar to the way we finish the last few iterations with loop unrolling.
- You should write your code so that it will compile and run correctly on any machine regardless of the value of $K$. Make use of the operation `sizeof` to do this.
- On some machines, unaligned writes can be much slower than aligned ones. (On some non-x86 machines, they can even cause segmentation faults.) Write your code so that it starts with byte-level writes until the destination address is a multiple of $K$, then do word-level writes, and then (if necessary) finish with byte-level writes.
- Beware of the case where `cnt` is small enough that the upper bounds on some of the loops become negative. With expressions involving the `sizeof` operator, the testing may be performed with unsigned arithmetic. (See Section 2.2.8 and Problem 2.72.)

**5.18** ◆◆◆

We considered the task of polynomial evaluation in Practice Problems 5.5 and 5.6, with both a direct evaluation and an evaluation by Horner's method. Try to write

faster versions of the function using the optimization techniques we have explored, including loop unrolling, parallel accumulation, and reassociation. You will find many different ways of mixing together Horner's scheme and direct evaluation with these optimization techniques.

Ideally, you should be able to reach a CPE close to the throughput limit of your machine. Our best version achieves a CPE of 1.07 on our reference machine.

### 5.19 ◆◆◆

In Problem 5.12, we were able to reduce the CPE for the prefix-sum computation to 3.00, limited by the latency of floating-point addition on this machine. Simple loop unrolling does not improve things.

Using a combination of loop unrolling and reassociation, write code for a prefix sum that achieves a CPE less than the latency of floating-point addition on your machine. Doing this requires actually increasing the number of additions performed. For example, our version with two-way unrolling requires three additions per iteration, while our version with four-way unrolling requires five. Our best implementation achieves a CPE of 1.67 on our reference machine.

Determine how the throughput and latency limits of your machine limit the minimum CPE you can achieve for the prefix-sum operation.

## Solutions to Practice Problems

### Solution to Problem 5.1 (page 536)

This problem illustrates some of the subtle effects of memory aliasing.

As the following commented code shows, the effect will be to set the value at `xp` to zero:

```
4       *xp = *xp + *xp; /* 2x */
5       *xp = *xp - *xp; /* 2x-2x = 0 */
6       *xp = *xp - *xp; /* 0-0 = 0 */
```

This example illustrates that our intuition about program behavior can often be wrong. We naturally think of the case where `xp` and `yp` are distinct but overlook the possibility that they might be equal. Bugs often arise due to conditions the programmer does not anticipate.

### Solution to Problem 5.2 (page 540)

This problem illustrates the relationship between CPE and absolute performance. It can be solved using elementary algebra. We find that for $n \leq 2$, version 1 is the fastest. Version 2 is fastest for $3 \leq n \leq 7$, and version 3 is fastest for $n \geq 8$.

### Solution to Problem 5.3 (page 548)

This is a simple exercise, but it is important to recognize that the four statements of a `for` loop—initial, test, update, and body—get executed different numbers of times.

| Code | min | max | incr | square |
|---|---|---|---|---|
| A. | 1 | 91 | 90 | 90 |
| B. | 91 | 1 | 90 | 90 |
| C. | 1 | 1 | 90 | 90 |

### Solution to Problem 5.4 (page 552)

This assembly code demonstrates a clever optimization opportunity detected by GCC. It is worth studying this code carefully to better understand the subtleties of code optimization.

A. In the less optimized code, register `%xmm0` is simply used as a temporary value, both set and used on each loop iteration. In the more optimized code, it is used more in the manner of variable `acc` in `combine4`, accumulating the product of the vector elements. The difference with `combine4`, however, is that location `dest` is updated on each iteration by the second `vmovsd` instruction.

We can see that this optimized version operates much like the following C code:

```
1  /* Make sure dest updated on each iteration */
2  void combine3w(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      data_t *data = get_vec_start(v);
7      data_t acc = IDENT;
8
9      /* Initialize in event length <= 0 */
10     *dest = acc;
11
12     for (i = 0; i < length; i++) {
13         acc = acc OP data[i];
14         *dest = acc;
15     }
16 }
```

B. The two versions of `combine3` will have identical functionality, even with memory aliasing.

C. This transformation can be made without changing the program behavior, because, with the exception of the first iteration, the value read from `dest` at the beginning of each iteration will be the same value written to this register

at the end of the previous iteration. Therefore, the combining instruction can simply use the value already in `%xmm0` at the beginning of the loop.

### Solution to Problem 5.5 (page 566)

Polynomial evaluation is a core technique for solving many problems. For example, polynomial functions are commonly used to approximate trigonometric functions in math libraries.

A. The function performs $2n$ multiplications and $n$ additions.

B. We can see that the performance-limiting computation here is the repeated computation of the expression `xpwr = x * xpwr`. This requires a floating-point multiplication (5 clock cycles), and the computation for one iteration cannot begin until the one for the previous iteration has completed. The updating of `result` only requires a floating-point addition (3 clock cycles) between successive iterations.

### Solution to Problem 5.6 (page 566)

This problem demonstrates that minimizing the number of operations in a computation may not improve its performance.

A. The function performs $n$ multiplications and $n$ additions, half the number of multiplications as the original function `poly`.

B. We can see that the performance-limiting computation here is the repeated computation of the expression `result = a[i] + x*result`. Starting from the value of `result` from the previous iteration, we must first multiply it by `x` (5 clock cycles) and then add it to `a[i]` (3 cycles) before we have the value for this iteration. Thus, each iteration imposes a minimum latency of 8 cycles, exactly our measured CPE.

C. Although each iteration in function `poly` requires two multiplications rather than one, only a single multiplication occurs along the critical path per iteration.

### Solution to Problem 5.7 (page 568)

The following code directly follows the rules we have stated for unrolling a loop by some factor $k$:

```
1  void unroll5(vec_ptr v, data_t *dest)
2  {
3      long i;
4      long length = vec_length(v);
5      long limit = length-4;
6      data_t *data = get_vec_start(v);
7      data_t acc = IDENT;
8
```

```
9       /* Combine 5 elements at a time */
10      for (i = 0; i < limit; i+=5) {
11          acc = acc OP data[i]   OP data[i+1];
12          acc = acc OP data[i+2] OP data[i+3];
13          acc = acc OP data[i+4];
14      }
15
16      /* Finish any remaining elements */
17      for (; i < length; i++) {
18          acc = acc OP data[i];
19      }
20      *dest = acc;
21  }
```

### Solution to Problem 5.8 (page 581)

This problem demonstrates how small changes in a program can yield dramatic performance differences, especially on a machine with out-of-order execution. Figure 5.39 diagrams the three multiplication operations for a single iteration of the function. In this figure, the operations shown as blue boxes are along the critical path—they need to be computed in sequence to compute a new value for loop variable `r`. The operations shown as light boxes can be computed in parallel with the critical path operations. For a loop with $P$ operations along the critical path, each iteration will require a minimum of $5P$ clock cycles and will compute the product for three elements, giving a lower bound on the CPE of $5P/3$. This implies lower bounds of 5.00 for A1, 3.33 for A2 and A5, and 1.67 for A3 and A4. We ran these functions on an Intel Core i7 Haswell processor and found that it could achieve these CPE values.

### Solution to Problem 5.9 (page 589)

This is another demonstration that a slight change in coding style can make it much easier for the compiler to detect opportunities to use conditional moves:

```
while (i1 < n && i2 < n) {
    long v1 = src1[i1];
```

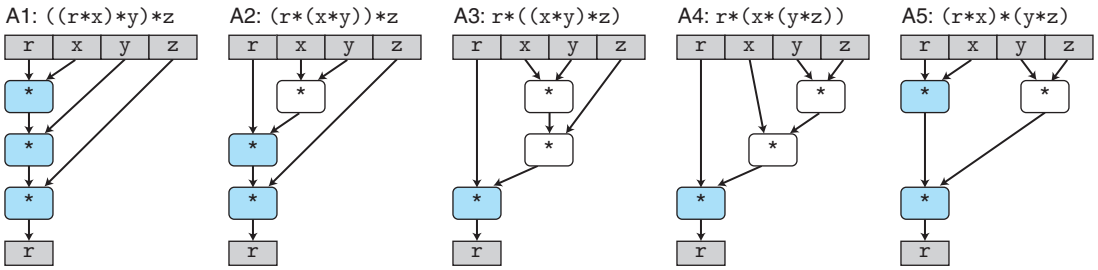



**Figure 5.39 Data dependencies among multiplication operations for cases in Problem 5.8.** The operations shown as blue boxes form the critical paths for the iterations.

```
    long v2 = src2[i2];
    long take1 = v1 < v2;
    dest[id++] = take1 ? v1 : v2;
    i1 += take1;
    i2 += (1-take1);
}
```

We measured a CPE of around 12.0 for this version of the code, a modest improvement over the original CPE of 15.0.

### Solution to Problem 5.10 (page 595)

This problem requires you to analyze the potential load-store interactions in a program.

A. It will set each element `a[`$i$`]` to $i + 1$, for $0 \leq i \leq 998$.

B. It will set each element `a[`$i$`]` to 0, for $1 \leq i \leq 999$.

C. In the second case, the load of one iteration depends on the result of the store from the previous iteration. Thus, there is a write/read dependency between successive iterations.

D. It will give a CPE of 1.2, the same as for Example A, since there are no dependencies between stores and subsequent loads.

### Solution to Problem 5.11 (page 597)

We can see that this function has a write/read dependency between successive iterations—the destination value `p[i]` on one iteration matches the source value `p[i-1]` on the next. A critical path is therefore formed for each iteration consisting of a store (from the previous iteration), a load, and a floating-point addition. The CPE measurement of 9.0 is consistent with our measurement of 7.3 for the CPE of `write_read` when there is a data dependency, since `write_read` involves an integer addition (1 clock-cycle latency), while `psum1` involves a floating-point addition (3 clock-cycle latency).

### Solution to Problem 5.12 (page 597)

Here is a revised version of the function:

```
1   void psum1a(float a[], float p[], long n)
2   {
3       long i;
4       /* last_val holds p[i-1]; val holds p[i] */
5       float last_val, val;
6       last_val = p[0] = a[0];
7       for (i = 1; i < n; i++) {
8           val  = last_val + a[i];
9           p[i] = val;
10          last_val = val;
11      }
12  }
```

We introduce a local variable `last_val`. At the start of iteration `i`, it holds the value of `p[i-1]`. We then compute `val` to be the value of `p[i]` and to be the new value for `last_val`.

This version compiles to the following assembly code:

```
   Inner loop of psum1a
   a in %rdi, i in %rax, cnt in %rdx, last_val in %xmm0
1  .L16:                                          loop:
2    vaddss  (%rdi,%rax,4), %xmm0, %xmm0            last_val = val = last_val + a[i]
3    vmovss  %xmm0, (%rsi,%rax,4)                   Store val in p[i]
4    addq    $1, %rax                               Increment i
5    cmpq    %rdx, %rax                             Compare i:cnt
6    jne     .L16                                   If !=, goto loop
```

This code holds `last_val` in `%xmm0`, avoiding the need to read `p[i-1]` from memory and thus eliminating the write/read dependency seen in `psum1`.