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 4.0×10^9 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