Chapter 5

Optimizing Program Performance

Writing an efficient program requires two types of activities. First, we must select the best 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 to optimize. Programmers can often write their programs in ways that make it easier for compilers to generate efficient code.

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 will run. 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 a program that will be run only once to generate a set of data points, it is more important to write it in a way that minimizes programming effort and ensures correctness. For code that will be executed repeatedly in a performance-critical environment, such as in a network router, much more extensive optimization usually is appropriate.

In this chapter, 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. In reality, compilers can only perform limited transformations of the program, and they can be thwarted by *optimization blockers*—aspects of the program's behavior that depend strongly on the execution environment. Programmers must assist the compiler by writing code that can be optimized readily. In the compiler literature, optimization techniques are classified as either "machine independent," which means that they should be applied regardless of the characteristics of the computer that will execute the code, or as "machine dependent," which means they depend on many low-level details of the machine. We organize our presentation along similar lines, starting with program transformations that should be standard practice when writing any program. We then progress to transformations whose efficacy depends on the characteristics of the target machine and compiler. These transformations also tend to reduce

the modularity and readability of the code and thus should be applied when maximum performance is the dominant concern.

To maximize the performance of a program, both the programmer and the compiler need to have 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 some recent models of Intel processors. We also devise a graphical notation that can be used to visualize the execution of instructions by the processor and to predict program performance.

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 we should focus on in our optimization efforts. Finally, we present an important observation, known as *Amdahl's law*, which quantifies the overall effect of optimizing some portion of a system.

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, while 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 is one of the most effective means of gaining some 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. One can identify performance-reducing attributes, such as excessive memory references and poor use of registers. Starting with the assembly code, we can even predict what operations will be performed in parallel and how well they will use the processor resources.

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. The ability of compilers to optimize programs is limited by several factors, including: (1) the requirement that they should never alter correct program behavior, (2) their limited understanding of the program's behavior and the environment in which it will be used, and (3) the need to perform the optimizations quickly.

Compiler optimization is supposed to be invisible to the user. When a programmer compiles code with optimization enabled (e.g., using the `-O` command line option), the code should have identical behavior as when compiled otherwise, except that it should run faster. This requirement restricts the ability of the

compiler to perform some types of optimizations.

Consider, for example, the following two procedures:

```
1 void twiddle1(int *xp, int *yp)
2 {
3     *xp += *yp;
4     *xp += *yp;
5 }
6
7 void twiddle2(int *xp, int *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. Therefore it cannot generate code in the style of `twiddle2` as an optimized version of `twiddle1`.

This phenomenon is known as *memory aliasing*. The compiler must assume that different pointers may designate a single place in memory. 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.

Practice Problem 5.1:

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(int *xp, int *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 int f(int);
2
3 int func1(x)
4 {
5     return f(x) + f(x) + f(x) + f(x);
6 }
7
8 int func2(x)
9 {
10    return 4*f(x);
11 }
```

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 int counter = 0;
2
3 int f(int x)
4 {
5     return counter++;
6 }
```

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 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 0.

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 all function calls intact.

Among compilers, the GNU compiler 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.

code/opt/vsum.c

```
1  void vsum1(int n)
2  {
3      int i;
4
5      for (i = 0; i < n; i++)
6          c[i] = a[i] + b[i];
7  }
8
9  /* Sum vector of n elements (n must be even) */
10 void vsum2(int n)
11 {
12     int i;
13
14     for (i = 0; i < n; i+=2) {
15         /* Compute two elements per iteration */
16         c[i]   = a[i]   + b[i];
17         c[i+1] = a[i+1] + b[i+1];
18     }
19 }
```

code/opt/vsum.c

Figure 5.1: **Vector sum functions.** These provide examples for how we express program performance.

5.2 Expressing Program Performance

We need a way to express program performance that can guide us in improving the code. A useful measure for many programs is *cycles per element* (CPE). This measure helps us understand the loop performance of an iterative program at a detailed level. Such a measure 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, expressed in either *megahertz* (MHz), millions of cycles per second, or *gigahertz* (GHz), billions of cycles per second. For example, when product literature characterizes a system as a "1.4 GHz" processor, it means that the processor clock runs at 1400 megahertz. The time required for each clock cycle is given by the reciprocal of the clock frequency. These typically are expressed in *nanoseconds* (i.e., billionths of a second). A 2-GHz clock has a 0.5-nanosecond period, while a 500-MHz clock has a period of 2 nanoseconds. From a programmer's perspective, it is more instructive to express measurements in clock cycles rather than nanoseconds. That way, the measurements are less dependent on the particular model of processor being evaluated, and they help us understand exactly how the program is being executed by the machine.

Many procedures contain a loop that iterates over a set of elements. For example, functions `vsum1` and `vsum2` in Figure 5.1 both compute the sum of two vectors of length n. The first computes one element of the destination vector per iteration. The second uses a technique known as *loop unrolling* to compute two elements per iteration. This version will only work properly for even values of n. Later in this chapter we

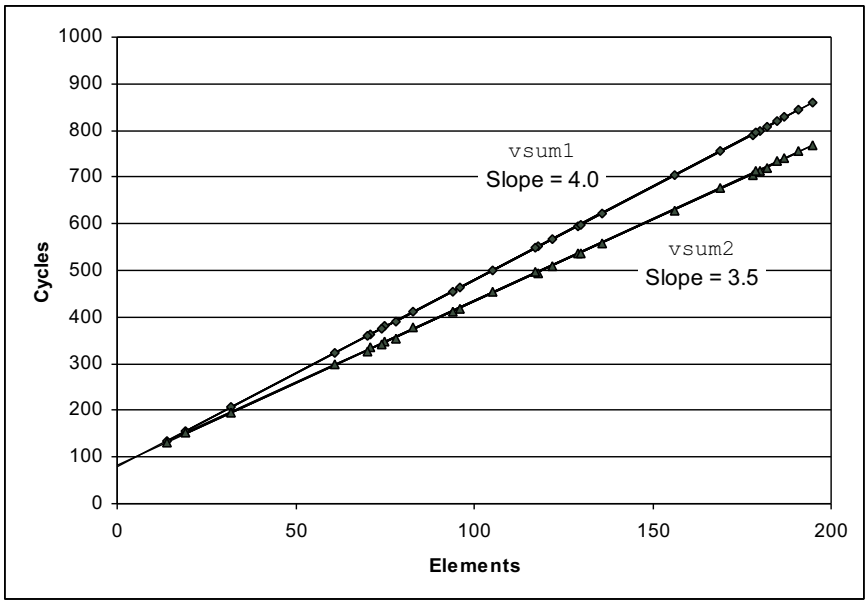


Figure 5.2: **Performance of vector sum functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

cover loop unrolling in more detail, including how to make it work for arbitrary values of n.

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 two function run times (in clock cycles) can be approximated by lines with equations $80 + 4.0n$ and $83.5 + 3.5n$, respectively. These equations indicated an overhead of 80 to 84 cycles to initiate the procedure, set up the loop, and complete the procedure, plus a linear factor of 3.5 or 4.0 cycles per element. For large values of n (say greater than 50), 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*, abbreviated "CPE." Note that 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, `vsum2`, with a CPE of 3.50, is superior to `vsum1`, with a CPE of 4.0.

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

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

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. **End Aside.**

Practice Problem 5.2:

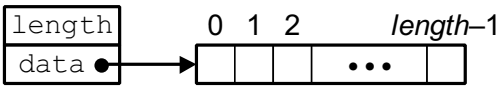


Figure 5.3: **Vector abstract data type.** A vector is represented by header information plus array of designated length.

Later in this chapter we will take 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, consider the simple vector data structure, shown in Figure 5.3. A vector is represented with two blocks of memory. The header is a structure declared as follows:

code/opt/vec.h

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

code/opt/vec.h

The declaration uses data type `data_t` to designate the data type of the underlying elements. In our evaluation, we measure the performance of our code for data types `int`, `float`, and `double`. We do this by compiling and running the program separately for different type declarations, as in the following example:

```
typedef int data_t;
```

In addition to the header, we allocate an array of `len` objects of type `data_t` to hold the actual vector elements.

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

code/opt/vec.c

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

.

code/opt/combine.c

```
1 /* Implementation with maximum use of data abstraction */
2 void combine1(vec_ptr v, data_t *dest)
3 {
4     int 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 OPER val;
11    }
12 }
```

code/opt/combine.c

Figure 5.5: **Initial implementation of combining operation.** Using different declarations of identity element *IDENT* and combining operation *OPER*, we can measure the routine for different operations.

in many other languages, including Java. Bounds checking reduces the chances of program error, but, as we will see, it also significantly affects program performance.

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 `OPER`, the code can be recompiled to perform different operations on the data. In particular, using the declarations

```
#define IDENT 0
#define OPER  +
```

it sums the elements of the vector. Using the declarations

```
#define IDENT 1
#define OPER  *
```

it computes the product of the vector elements.

As a starting point, here are the CPE measurements for `combine1` running on an Intel Pentium III, trying all combinations of data type and combining operation. In our measurements, we found that the timings were generally equal for single and double-precision floating point data. We therefore show only the measurements for single precision.

Function	Page	Method	Integer		Floating point	
			+	*	+	*
`combine1`	373	Abstract unoptimized	42.06	41.86	41.44	160.00
`combine1`	373	Abstract `-O2`	31.25	33.25	31.25	143.00

code/opt/combine.c

```
1 /* Move call to vec_length out of loop */
2 void combine2(vec_ptr v, data_t *dest)
3 {
4     int i;
5     int 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 OPER val;
12    }
13 }
```

code/opt/combine.c

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.

By default, the compiler generates code suitable for stepping with a symbolic debugger. Very little optimization is performed since the intention is to make the object code closely match the computations indicated in the source code. By simply setting the command line switch to '`-O2`' we enable optimizations. As can be seen, this significantly improves the program performance. In general, it is good to get into the habit of enabling this level of optimization, unless the program is being compiled with the intention of debugging it. For the remainder of our measurements, we enable this level of compiler optimization.

Note also that the times are fairly comparable for the different data types and the different operations, with the exception of floating-point multiplication. These very high cycle counts for multiplication are due to an anomaly in our benchmark data. Identifying such anomalies is an important component of performance analysis and optimization. We return to this issue in Section 5.11.1. We will see that we can improve on this performance considerably.

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 loops 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`, which calls `vec_length` at the beginning and assigns the result to a local variable `length`. This local variable is then used in the test condition of the for loop. Surprisingly, this small change significantly affects program performance. As the following table shows, we eliminate approximately 10 clock cycles for each vector element with this simple transformation:

Function	Page	Method	Integer		Floating point	
			+	*	+	*
`combine1`	373	Abstract `-O2`	31.25	33.25	31.25	143.00
`combine2`	374	Move `vec_length`	22.61	21.25	21.15	135.00

This optimization is an instance of a general class of optimizations known as *code motion*. They involve identifying a computation that is performed multiple 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. In cases such as these, the programmer must help the compiler by explicitly performing the 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 lower case. The procedure steps through the string, converting each uppercase character to lower case.

The library procedure `strlen` is called as part of the loop test of `lower1`. A simple version of `strlen` is also shown in Figure 5.7. Since strings in C are null-terminated character sequences, `strlen` must step 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 on each of the n iterations of `lower1`, the overall run time of `lower1` is quadratic in the string length.

This analysis is confirmed by actual measurements of the procedure for different length strings, as shown Figure 5.8. The graph of the run time for `lower1` rises steeply as the string length increases. The lower part of the figure shows the run times for eight 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 quadratic complexity. For a string of length 262,144, `lower1` requires a full 3.1 minutes of CPU 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 262,144, the function requires just 0.006 seconds—over 30,000 times faster than `lower1`. Each doubling of the string length causes a doubling of the run time—a clear indicator of linear complexity. For longer strings, the run time improvement will be even greater.

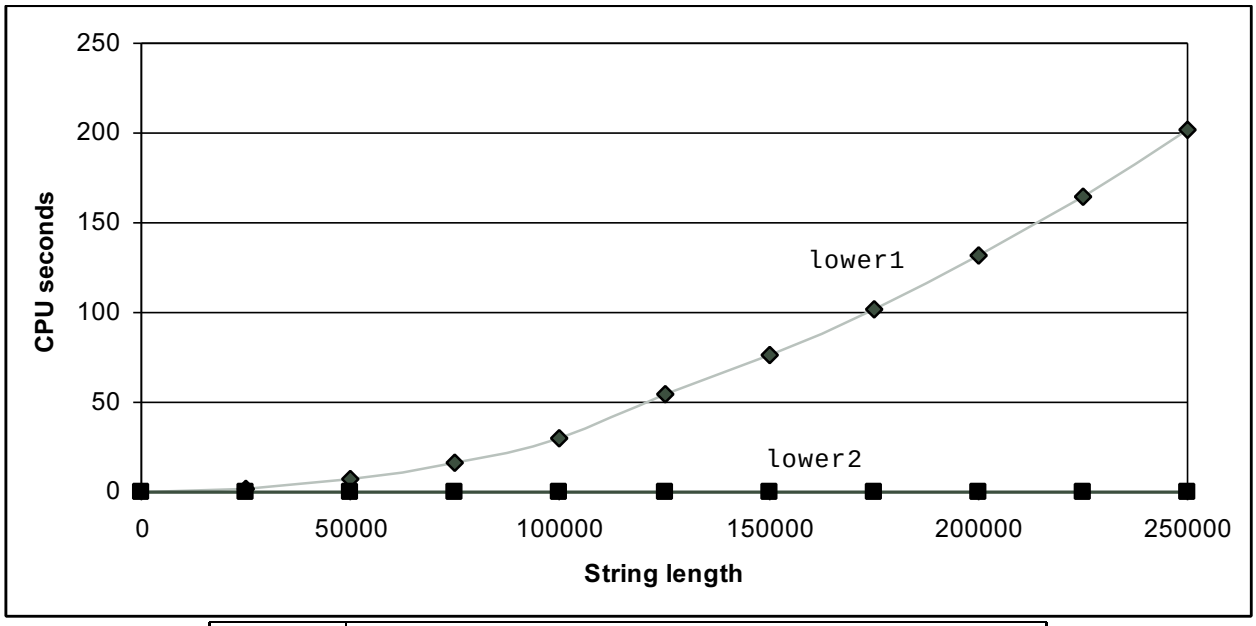
In an ideal world, a compiler would recognize that each call to `strlen` in the loop test will return the same result, and thus the call could be moved out of the loop. This would require a very sophisticated analysis, since `strlen` checks the elements of the string and these values are changing as `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 aggressive compilers, so programmers must do such transformations themselves.

code/opt/lower.c

```
1  /* Convert string to lower case: slow */
2  void lower1(char *s)
3  {
4      int 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 lower case: faster */
12 void lower2(char *s)
13 {
14     int i;
15     int 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 /* Implementation of library function strlen */
23 /* Compute length of string */
24 size_t strlen(const char *s)
25 {
26     int length = 0;
27     while (*s != '\0') {
28         s++;
29         length++;
30     }
31     return length;
32 }
```

code/opt/lower.c

Figure 5.7: **Lower-case conversion routines.** The two procedures have radically different performance.



Function	String length					
	8,192	16,384	32,768	65,536	131,072	262,144
`lower1`	0.15	0.62	3.19	12.75	51.01	186.71
`lower2`	0.0002	0.0004	0.0008	0.0016	0.0031	0.0060

Figure 5.8: **Comparative performance of lower-case conversion routines.** The original code `lower1` has quadratic asymptotic complexity due to an inefficient loop structure. The modified code `lower2` has linear complexity.

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 lower-case 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 a string of one million characters, for which `lower1` would over require nearly one hour of CPU time. All of a sudden this benign piece of code has become a major performance bottleneck. By contrast, `lower2` would complete in well under a second. 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:

Consider the following functions:

```
int min(int x, int y) { return x < y ? x : y; }
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int square(int 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.
```
int low = min(x, y);
int 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 incur substantial overhead and 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 procedure is especially costly since it performs bounds checking. 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

code/opt/combine.c

```
1 /* Direct access to vector data */
2 void combine3(vec_ptr v, data_t *dest)
3 {
4     int i;
5     int 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 OPER data[i];
11    }
12 }
```

code/opt/combine.c

Figure 5.9: **Eliminating function calls within the loop.** The resulting code runs much faster, at some cost in program modularity.

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 the advantage of this transformation on the basis of the following experimental results:

Function	Page	Method	Integer		Floating point	
			+	*	+	*
`combine2`	374	Move `vec_length`	20.66	21.25	21.15	135.00
`combine3`	379	Direct data access	6.00	9.00	8.00	117.00

There is a improvement of up to a factor of 3.5X. For applications in which performance is a significant issue, one often must compromise modularity and abstraction for speed. It is wise to include documentation on the transformations applied, as well as the as assumptions that led to them, in case the code needs to be modified later.

Aside: Expressing relative performance.
The best way to express a performance improvement is as a ratio of the form T_{old}/T_{new}, where T_{old} is the time required for the original version and T_{new} is the time required by the modified version. This will be a number greater than 1.0 if any real improvement occurred. We use the suffix 'X' to indicate such a ratio, where the factor "3.5X" is expressed verbally as "3.5 times."

The more traditional way of expressing relative change as a percentage works well when the change is small, but its definition is ambiguous. Should it be $100 \cdot (T_{old} - T_{new})/T_{new}$ or possibly $100 \cdot (T_{old} - T_{new})/T_{old}$, or something else? In addition, it is less instructive for large changes. Saying that "performance improved by 250%" is more difficult to comprehend than simply saying that the performance improved by a factor of 3.5. **End Aside.**

5.6 Eliminating Unneeded Memory References

The code for `combine3` accumulates the value being computed by the combining operation at the location designated by pointer `dest`. This attribute can be seen by examining the assembly code generated for the compiled loop, with integers as the data type and multiplication as the combining operation. In this code, register `%ecx` points to `data`, `%edx` contains the value of `i`, and `%edi` points to `dest`.

```
    combine3: type=INT, OPER = *
    dest in %edi, data in %ecx, i in %edx, length in %esi
1 .L18:                                  loop:
2   movl (%edi),%eax                       Read *dest
3   imull (%ecx,%edx,4),%eax               Multiply by data[i]
4   movl %eax,(%edi)                       Write *dest
5   incl %edx                              i++
6   cmpl %esi,%edx                         Compare i:length
7   jl .L18                                If <, goto loop
```

code/opt/combine.c

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

code/opt/combine.c

Figure 5.10: **Accumulating result in temporary.** This eliminates the need to read and write intermediate values on every loop iteration.

Instruction 2 reads the value stored at `dest` and instruction 4 writes back to this location. This seems wasteful, since the value read by instruction 2 on the next iteration normally will be the value that has just been written.

This leads to the optimization shown as `combine4` in Figure 5.10, where we introduce a temporary variable `x` 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 `%eax` 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. Registers `%ecx` and `%edx` are used as before, but there is no need to reference `*dest`.

```
    combine4: type=INT, OPER = *
    data in %eax, x in %ecx, i in %edx, length in %esi
1 .L24:                                 loop:
2   imull (%eax,%edx,4),%ecx              Multiply x by data[i]
3   incl %edx                             i++
4   cmpl %esi,%edx                        Compare i:length
5   jl .L24                               If <, goto loop
```

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

Function	Page	Method	Integer		Floating point	
			+	*	+	*
`combine3`	379	Direct data access	6.00	9.00	8.00	117.00
`combine4`	381	Accumulate in temporary	2.00	4.00	3.00	5.00

The most dramatic decline is in the time for floating-point multiplication. Its time becomes comparable to the times for the other combinations of data type and operation. We will examine the cause for this sudden

decrease in Section 5.11.1.

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 behavior 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` be a vector consisting of the three elements $[2, 3, 5]$ 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 final 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, an optimizing compiler cannot make a judgement about the conditions under which a function might be used and what the programmer's intentions might be. Instead, when given `combine3` to compile, it is obligated to preserve its exact functionality, even if this means generating inefficient code.

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 begin to consider optimizations that make more use of the means by which processors execute instructions and the capabilities of particular processors. 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 simple operational model of how modern processors work. 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 assembly-language

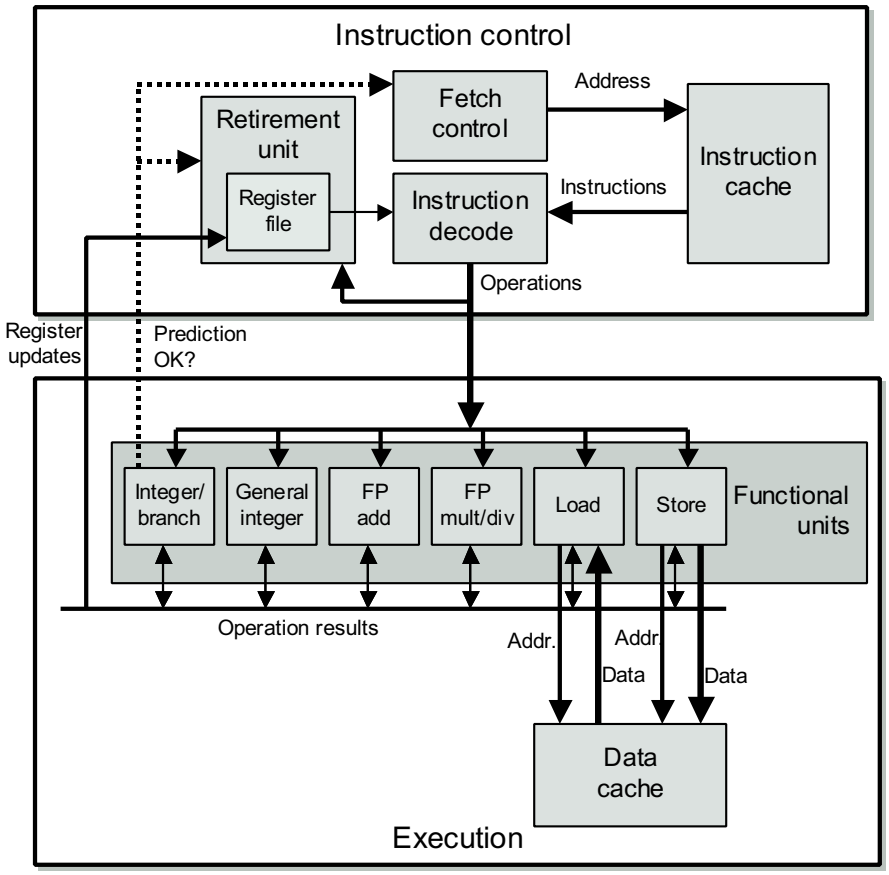


Figure 5.11: **Block diagram of a modern processor.** The Instruction control unit is responsible for reading instructions from memory and generating a sequence of primitive operations. The Execution unit then performs the operations and indicates whether the branches were correctly predicted.

programs. At the assembly-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. In some designs, there can be 80 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.

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 Intel "P6" microarchitecture [30], the basis for the Intel PentiumPro, Pentium II and Pentium III processors. The newer Pentium 4 has a different microarchitecture, but it has a similar overall structure to the one we present here. The P6 microarchitecture typifies the high-end processors produced by a number of manufacturers since the late 1990s. It is described in the industry as being *superscalar*, which

means it 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 assembly 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.

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 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. A more exotic technique would be to begin fetching and executing instructions for both possible directions, later discarding the results for the incorrect direction. To date, this approach has not been considered cost effective. 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*. 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 an IA32 processor, an instruction can be decoded into a variable number of operations. The details vary from one processor design to another, but we attempt to describe a typical implementation. In this machine, decoding the instruction

```
addl %eax,%edx
```

yields a single addition operation, whereas decoding the instruction

```
addl %eax,4(%edx)
```

yields 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. This 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. For machines with simple instructions, the operations correspond more closely to the original instructions.

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 specific types of operations. Our figure illustrates a typical set of functional units. It is styled after those found in recent Intel processors. The units in the figure are as follows:

[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.

Integer/Branch: Performs simple integer operations (add, test, compare, logical). Also processes branches, as is discussed below.

General Integer: Can handle all integer operations, including multiplication and division.

Floating-Point Add: Handles simple floating-point operations (addition, format conversion).

Floating-Point Multiplication/Division: Handles floating-point multiplication and division. More complex floating-point instructions, such transcendental functions, are converted into sequences of operations.

Load: Handles operations that read data from the memory into the processor. The functional unit has an adder to perform address computations.

Store: Handles operations that write data from the processor to the memory. The functional unit 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 to 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. 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 execution units. We explore this further in Section 5.12.

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 and floating-point 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 in 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 *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.

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

Operation	Latency	Issue time
Integer add	1	1
Integer multiply	4	1
Integer divide	36	36
Floating-point add	3	1
Floating-point multiply	5	2
Floating-point divide	38	38
Load (cache hit)	3	1
Store (cache hit)	3	1

Figure 5.12: **Performance of Pentium III arithmetic operations.** Latency represents the total number of cycles for a single operation. Issue time denotes the number of cycles between successive, independent operations. (Obtained from Intel literature).

giving a unique identifier to the result of the operation. An entry (r, t) is added to a table maintaining the association between each program register and the tag 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. By this mechanism, values can be passed directly from one operation to another, rather than being written to and read from the register file. 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.

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 ten 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. **End Aside.**

5.7.2 Functional Unit Performance

Figure 5.12 documents the performance of some of the basic operations for an Intel Pentium III. These timings are typical for other processors as well. Each operation is characterized by two cycle counts: the *latency*, which indicates the total number of cycles the functional unit requires to complete the operation; and the *issue time*, which indicates the number of cycles between successive independent operations. The latencies range from one cycle for basic integer operations, to several cycles for loads, stores, integer multiplication, and the more common floating-point operations, to many cycles for division and other complex

operations.

As the third column in Figure 5.12 shows, several functional units of the processor are *pipelined*, meaning that they can start on a new operation before the previous one is completed. The issue time indicates the number of cycles between successive operations for the unit. In a pipelined unit, the issue time is smaller than the latency. 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: one to process the exponent values, one to add the fractions, and one to round the final result. The 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. As indicated, most of the units can begin a new operation on every clock cycle. The only exceptions are the floating-point multiplier, which requires a minimum of two cycles between successive operations, and the two dividers, which are not pipelined at all.

Circuit designers can create functional units with a range of performance characteristics. Creating a unit with short latency or issue time 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, the 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 design of the Pentium III, 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 short latency or issue time, and so these operations are relatively slow.

5.7.3 A Closer Look at Processor Operation

As a tool for analyzing the performance of a machine level program executing on a modern processor, we have developed a more detailed textual notation to describe the operations generated by the instruction decoder, as well as a graphical notation to show the processing of operations by the functional units. Neither of these notations exactly represents the implementation of a specific, real-life processor. They are simply methods to help understand how a processor can take advantage of parallelism and branch prediction when executing a program.

Translating Instructions into Operations

We present our notation by working with `combine4` (Figure 5.10), our fastest code up to this point 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 cases of integer data with both multiplication and addition as the combining operations. The compiled code for this loop with multiplication consists of four instructions. In this code, register `%eax` holds the pointer `data`, `%edx` holds `i`, `%ecx` holds `x`, and `%esi` holds `length`:

```
combine4: type=INT, OPER = *
data in %eax, x in %ecx, i in %edx, length in %esi
```

```
1 .L24:                                  loop:
2   imull (%eax,%edx,4),%ecx               Multiply x by data[i]
3   incl %edx                              i++
4   cmpl %esi,%edx                         Compare i:length
5   jl .L24                                If <, goto loop
```

Every time the processor executes the loop, the instruction decoder translates these four instructions into a sequence of operations for the Execution unit. On the first iteration, with `i` equal to 0, our hypothetical machine would issue the following sequence of operations:

Assembly instructions	Execution unit operations
`.L24:`	
`imull (%eax,%edx,4),%ecx`	`load (%eax, %edx.0, 4)` → `t.1`
	`imull t.1, %ecx.0` → `%ecx.1`
`incl %edx`	`incl %edx.0` → `%edx.1`
`cmpl %esi,%edx`	`cmpl %esi, %edx.1` → `cc.1`
`jl .L24`	`jl-taken cc.1`

In our translation, we have converted the memory reference by the multiply instruction into an explicit `load` instruction that reads the data from memory into the processor. We also have assigned *operand labels* to the values that change each iteration. These labels are a stylized version of the tags generated by register renaming. Thus, the value in register `%ecx` is identified by the label `%ecx.0` at the beginning of the loop, and by `%ecx.1` after it has been updated. The register values that do not change from one iteration to the next would be obtained directly from the register file during decoding. We also introduce the label `t.1` to denote the value read by the `load` operation and passed to the `imull` operation, and we explicitly show the destination of the operation. Thus, the pair of operations

```
load (%eax, %edx.0, 4) → t.1
imull t.1, %ecx.0      → %ecx.1
```

indicates that the processor first performs a `load` operation, computing the address using the value of `%eax` (which does not change during the loop), and the value stored in `%edx` at the start of the loop. This will yield a temporary value, which we label `t.1`. The multiply operation then takes this value and the value of `%ecx` at the start of the loop and produces a new value for `%ecx`. As this example illustrates, tags can be associated with intermediate values that are never written to the register file.

The operation

```
incl %edx.0 → %edx.1
```

indicates that the increment operation adds 1 to the value of `%edx` at the start of the loop to generate a new value for this register.

The operation

```
cmpl %esi, %edx.1 → cc.1
```

Execution unit operations		
`load (%eax, %edx.0, 4)`	→	`t.1`
`imull t.1, %ecx.0`	→	`%ecx.1`
`incl %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

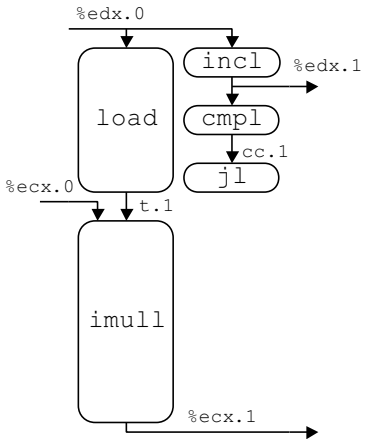


Figure 5.13: **Operations for first iteration of inner loop of `combine4` for integer multiplication.** Memory reads are explicitly converted to loads. Register names are tagged with instance numbers.

indicates that the compare operation (performed by either integer unit) compares the value in `%esi` (which does not change in the loop) with the newly computed value for `%edx`. It then sets the condition codes, identified with the explicit label `cc.1`. As this example illustrates, the processor can use renaming to track changes to the condition code registers.

Finally, the jump instruction was predicted as being taken. The jump operation

```
jl-taken cc.1
```

checks whether the newly computed values for the condition codes (`cc.1`) indicate this was the correct choice. If not, then it signals the ICU to begin fetching instructions at the instruction following the `jl`. To simplify the notation, we omit any information about the possible jump destinations. In practice, the processor must keep track of the destination for the unpredicted direction, so that it can begin fetching from there in the event the prediction is incorrect.

As this example translation shows, our operations mimic the structure of the assembly-language instructions in many ways, except that they refer to their source and destination operations by labels that identify different instances of the registers. In the actual hardware, register renaming dynamically assigns tags to indicate these different values. Tags are bit patterns rather than symbolic names such as "`%edx.1`," but they serve the same purpose.

Processing of Operations by the Execution Unit

Figure 5.13 shows the operations in two forms: that generated by the instruction decoder and that shown as a *computation graph* in which operations are represented by rounded boxes and arrows indicate the passing of data between operations. We only show the arrows for the operands that change from one iteration to the next, since only these values are passed directly between functional units.

The height of each operator box indicates how many cycles the operation requires—that is, the latency of

Execution unit operations		
`load (%eax, %edx.0, 4)`	→	`t.1`
`addl t.1, %ecx.0`	→	`%ecx.1`
`incl %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

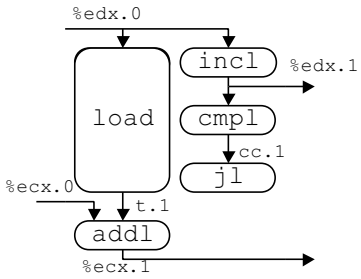


Figure 5.14: **Operations for first iteration of inner loop of `combine4` for integer addition.** Compared to multiplication, the only change is that the addition operation requires only one cycle.

that particular function. In this case, integer multiplication `imull` requires four cycles, load requires three, and the other operations require one. In demonstrating the timing of a loop, we position the blocks vertically to represent the times when operations are performed, with time increasing in the downward direction. We can see that the five operations for the loop form two parallel chains, indicating two series of computations that must be performed in sequence. The chain on the left processes the data, first reading an array element from memory and then multiplying it times the accumulated product. The chain on the right processes the loop index `i`, first incrementing it and then comparing it to `length`. The jump operation checks the result of this comparison to make sure the branch was correctly predicted. Note that there are no outgoing arrows from the jump operation box. If the branch was correctly predicted, no other processing is required. If the branch was incorrectly predicted, then the branch function unit will signal the instruction fetch control unit, and this unit will take corrective action. In either case, the other operations do not depend on the outcome of the jump operation.

Figure 5.14 shows the same translation into operations but with integer addition as the combining operation. As the graphical depiction shows, all of the operations, except load, now require just one cycle.

Scheduling of Operations with Unlimited Resources

To see how a processor would execute a series of iterations, imagine first a processor with an unlimited number of functional units and with perfect branch prediction. Each operation could then begin as soon as its data operands were available. The performance of such a processor would be limited only by the latencies and throughputs of the functional units, and the data dependencies in the program. Figure 5.15 shows the computation graph for the first three iterations of the loop in `combine4` with integer multiplication on such a machine. For each iteration, there is a set of five operations with the same configuration as those in Figure 5.13, with appropriate changes to the operand labels. The arrows from the operators of one iteration to those of another show the data dependencies between the different iterations.

Each operator is placed vertically at the highest position possible, subject to the constraint that no arrows can point upward, since this would indicate information flowing backward in time. Thus, the `load` operation of one iteration can begin as soon as the `incl` operation of the previous iteration has generated an updated value of the loop index.

The computation graph shows the parallel execution of operations by the Execution unit. On each cycle,

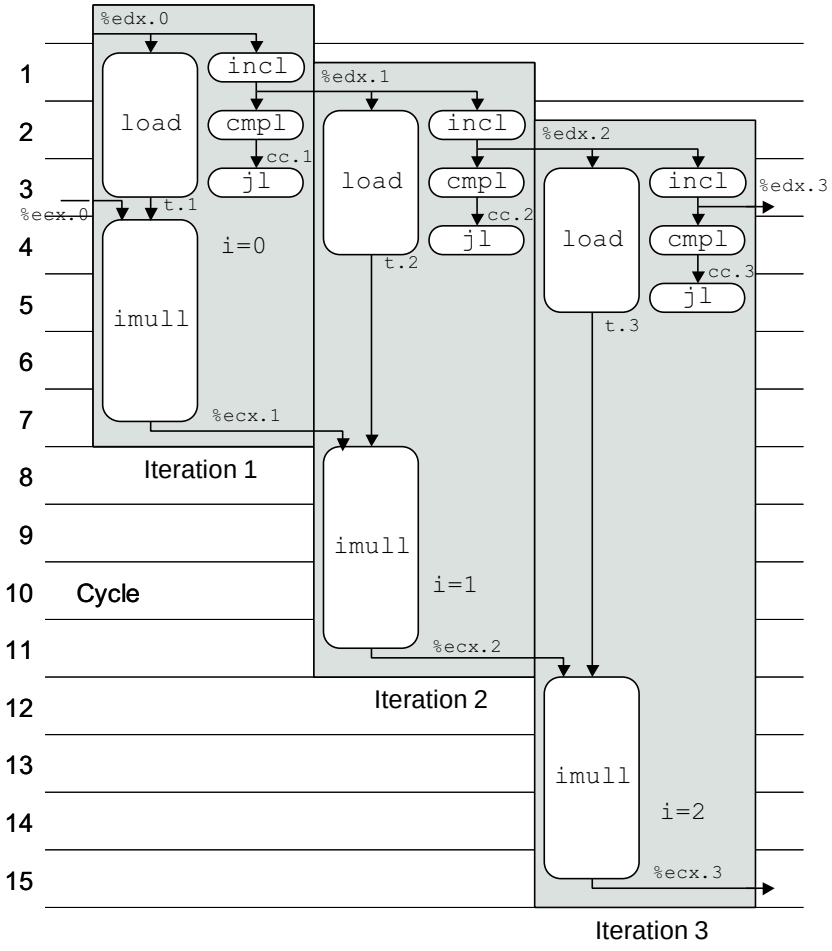


Figure 5.15: **Scheduling of operations for integer multiplication with unlimited number of execution units.** The 4 cycle latency of the multiplier is the performance-limiting resource.

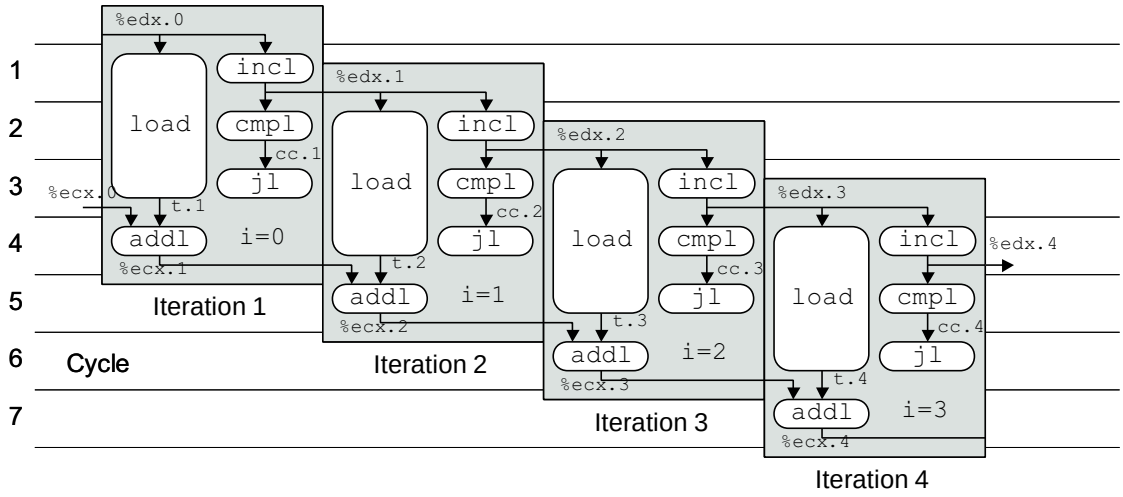


Figure 5.16: **Scheduling of operations for integer addition with unbounded resource constraints.** With unbounded resources the processor could achieve a CPE of 1.0.

all of the operations on one horizontal line of the graph execute in parallel. The graph also demonstrates out-of-order, speculative execution. For example, the `incl` operation in one iteration is executed before the `jl` instruction of the previous iteration has even begun. We can also see the effect of pipelining. Each iteration requires at least seven cycles from start to end, but successive iterations are completed every four cycles. Thus, the effective processing rate is one iteration every four cycles, giving a CPE of 4.0.

The four-cycle latency of integer multiplication constrains the performance of the processor for this program. Each `imull` operation must wait until the previous one has completed, since it needs the result of this multiplication before it can begin. In our figure, the multiplication operations begin on cycles 4, 8, and 12. With each succeeding iteration, a new multiplication begins every fourth cycle.

Figure 5.16 shows the first four iterations of `combine4` for integer addition on a machine with an unbounded number of functional units. With a single-cycle combining operation, the program could achieve a CPE of 1.0. We see that as the iterations progress, the Execution unit would perform parts of seven operations on each clock cycle. For example, in cycle 4 we can see that the machine is executing the `addl` for iteration 1; different parts of the `load` operations for iterations 2, 3, and 4; the `jl` for iteration 2; the `cmpl` for iteration 3; and the `incl` for iteration 4.

Scheduling of Operations with Resource Constraints

Of course, a real processor has only a fixed set of functional units. Unlike our earlier examples, where the performance was constrained only by the data dependencies and the latencies of the functional units, performance becomes limited by resource constraints as well. In particular, our processor has only two units capable of performing integer and branch operations. In contrast, the graph of Figure 5.15 has three of these operations in parallel on cycles 3 and four in parallel on cycle 4.

Figure 5.17 shows the scheduling of the operations for `combine4` with integer multiplication on a resource-constrained processor. We assume that the general integer unit and the branch/integer unit can each begin

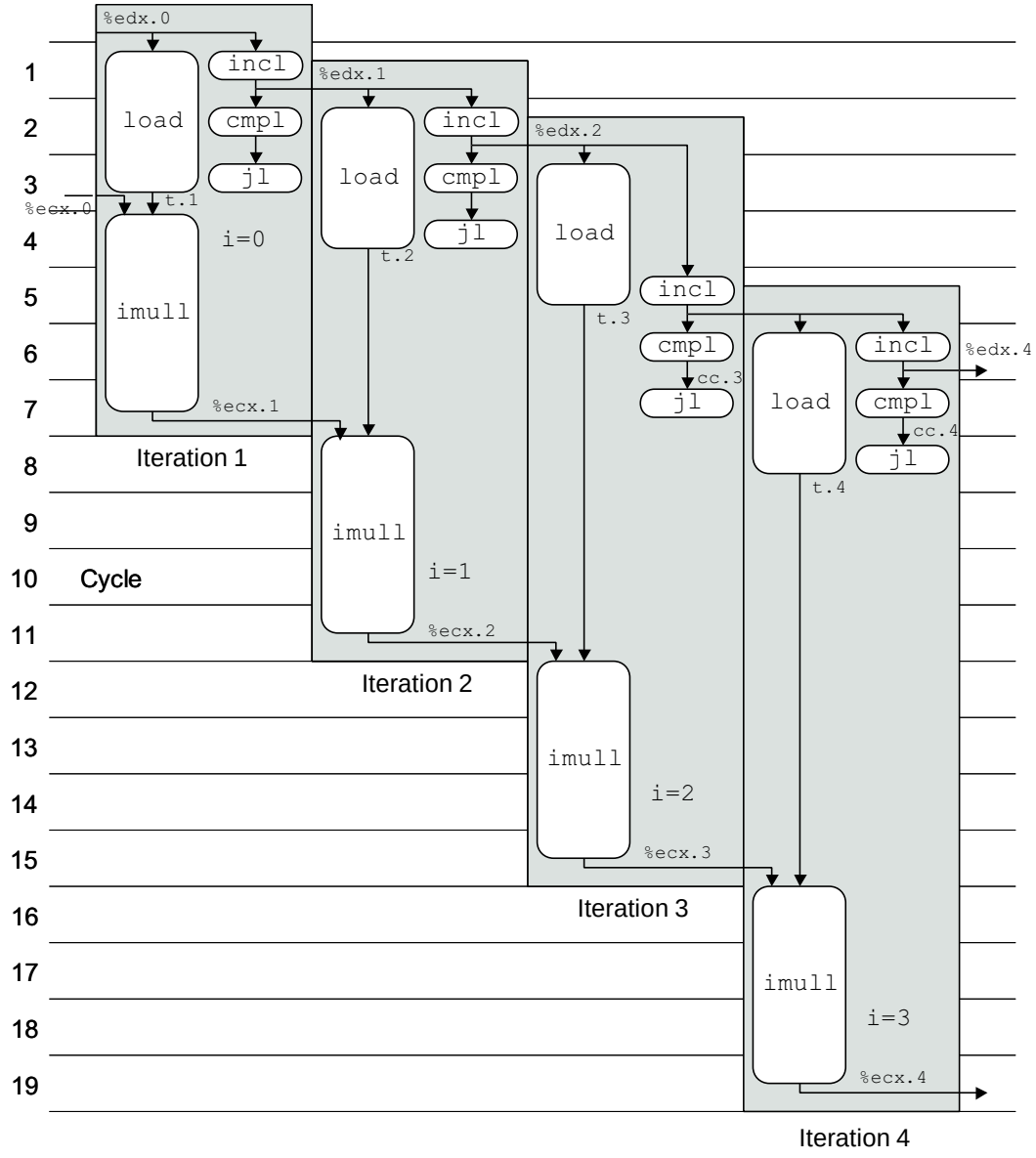


Figure 5.17: **Scheduling of operations for integer multiplication with actual resource constraints.** The multiplier latency remains the performance-limiting factor.

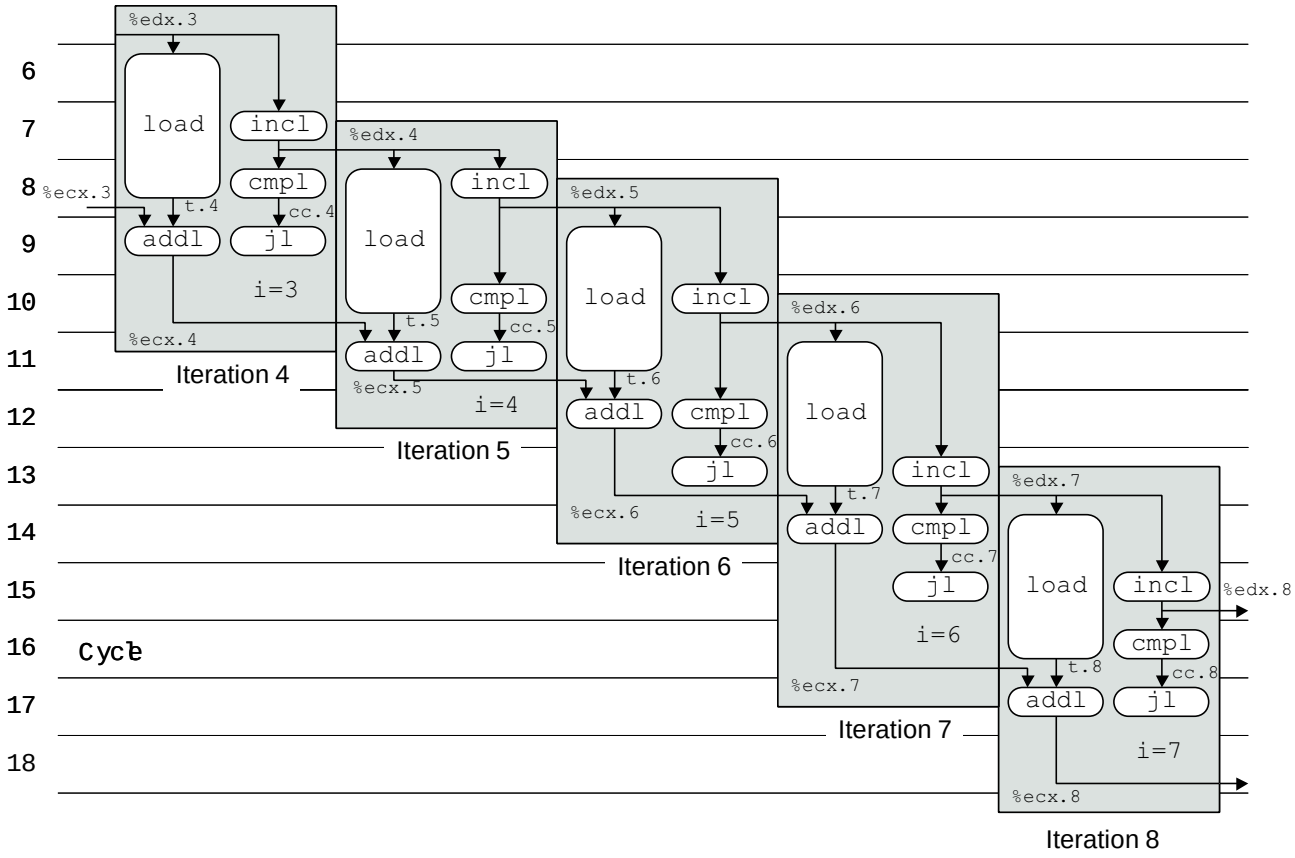


Figure 5.18: **Scheduling of operations for integer addition with actual resource constraints.** The limitation to two integer units constrains performance to a CPE of 2.0.

a new operation on every clock cycle. It is possible to have more than two integer or branch operations executing in parallel, as shown in cycle 6, because the `imull` operation is in its third cycle by this point.

With constrained resources, our processor must have some *scheduling policy* that determines which operation to perform when it has more than one choice. For example, in cycle 3 of the graph of Figure 5.15, we show three integer operations being executed: the `jl` of iteration 1, the `cmpl` of iteration 2, and the `incl` of iteration 3. For Figure 5.17, we must delay one of these operations. We do so by keeping track of the *program order* for the operations, that is, the order in which the operations would be performed if we executed the machine-level program in strict sequence. We then give priority to the operations according to their program order. In this example, we would defer the `incl` operation, since any operation of iteration 3 is later in program order than those of iterations 1 and 2. Similarly, in cycle 4, we would give priority to the `imull` operation of iteration 1 and the `jl` of iteration 2 over that of the `incl` operation of iteration 3.

For this example, the limited number of functional units does not slow down our program. Performance is still constrained by the four-cycle latency of integer multiplication.

For the case of integer addition, the resource constraints impose a clear limitation on program performance. Each iteration requires four integer or branch operations, and there are only two functional units for these

operations. Thus, we cannot hope to sustain a processing rate any better than two cycles per iteration. In creating the graph for multiple iterations of `combine4` for integer addition, an interesting pattern emerges. Figure 5.18 shows the scheduling of operations for iterations 4 through 8. We chose this range of iterations because it shows a regular pattern of operation timings. Observe how the timing of all operations in iterations 4 and 8 is identical, except that the operations in iteration 8 occur eight cycles later. As the iterations proceed, the patterns shown for iterations 4 to 7 would keep repeating. Thus, we complete four iterations every eight cycles, achieving the optimum CPE of 2.0.

Summary of `combine4` Performance

We now can consider the measured performance of `combine4` for all four combinations of data type and combining operations:

Function	Page	Method	Integer		Floating point	
			+	*	+	*
`combine4`	381	Accumulate in temporary	2.00	4.00	3.00	5.00

With the exception of integer addition, these cycle times nearly match the latency for the combining operation, as shown in Figure 5.12. Our transformations to this point have reduced the CPE value to the point where the time for the combining operation becomes the limiting factor.

For the case of integer addition, we have seen that the limited number of functional units for branch and integer operations limits the achievable performance. With four such operations per iteration, and just two functional units, we cannot expect the program to go faster than 2 cycles per iteration.

In general, processor performance is limited by three types of constraints. First, the data dependencies in the program force some operations to delay until their operands have been computed. Since the functional units have latencies of one or more cycles, this places a lower bound on the number of cycles in which a given sequence of operations can be performed. Second, the resource constraints limit how many operations can be performed at any given time. We have seen that the limited number of functional units is one such resource constraint. Other constraints include the degree of pipelining by the functional units, as well as limitations of other resources in the ICU and the EU. For example, an Intel Pentium III can only decode three instructions on every clock cycle. Finally, the success of the branch prediction logic constrains the degree to which the processor can work far enough ahead in the instruction stream to keep the execution unit busy. Whenever a misprediction occurs, a significant delay occurs getting the processor restarted at the correct location.

5.8 Reducing Loop Overhead

The performance of `combine4` for integer addition is limited by the fact that each iteration contains four instructions, with only two functional units capable of performing them. Only one of these four instructions operates on the program data. The others are part of the loop overhead of computing the loop index and testing the loop condition.