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 the issue of 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. This makes the programs more susceptible to bugs and more difficult to modify or extend. For a program that will just be run 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 may be 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 whose behavior depends 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," meaning that they should be applied regardless of the characteristics of the computer that will execute the code, or as "machine dependent," meaning 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 hence 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 is should use a multiply instruction or some combinations 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 devise a graphical notation that can be used to visualize the execution of instructions on the processor and to predict program performance.

We conclude 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. Finally, we present an important observation, known as *Amdahl's Law* quantifying 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, 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. Unfortunately, optimizing compilers have limitations, due to constraints imposed on their behavior, to the limited understanding they have of the program's behavior and how it will be used, and to the requirement that they perform the compilation 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 where `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 source.

Consider, however, the following code for `f`

```
1 int counter = 0;
2
3 int f(int x)
4 {
5     return counter++;
6 }
7
```

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 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 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), i.e., millions of cycles per second, or *Gigahertz* (GHz), i.e., 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 1,400 Megahertz. The time required for each clock cycle is given by the reciprocal of the clock frequency. These are typically 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 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

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.

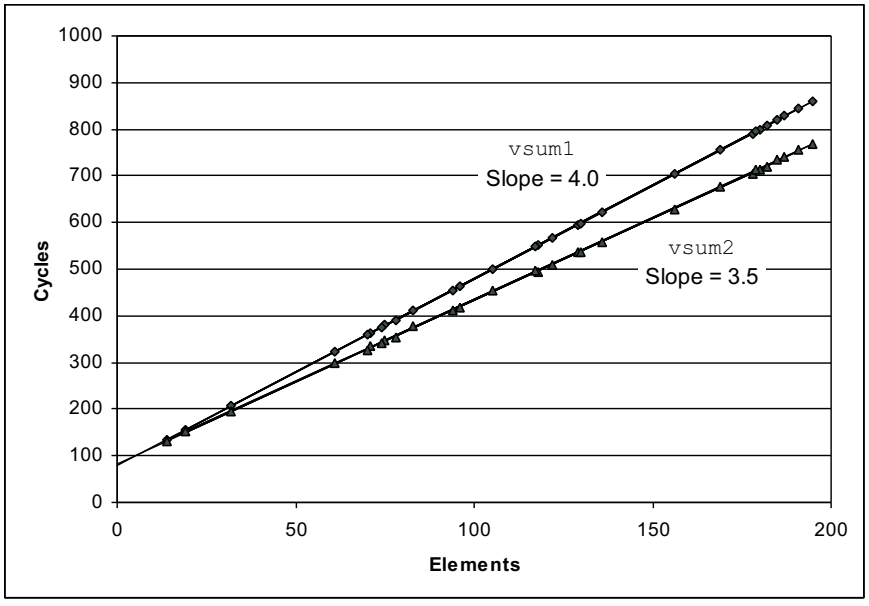


Figure 5.2: **Performance of Vector Sum Functions.** The slope of the lines indicates the number of clock cycles per element (CPE).

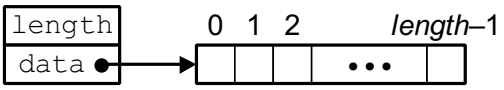


Figure 5.3: **Vector Abstract Data Type.** A vector is represented by header information plus array of designated length.

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.**

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, for 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 in many other languages, including Java. Bounds checking reduces the chances of program error, but, as we will see, 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

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.

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

we sum the elements of the vector. Using the declarations:

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

we compute 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`	213	Abstract unoptimized	42.06	41.86	41.44	160.00
`combine1`	213	Abstract `-O2`	31.25	33.25	31.25	143.00

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.

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.

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`, that 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 has a significant effect on program performance.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	213	Abstract `-O2`	31.25	33.25	31.25	143.00
`combine2`	214	Move `vec_length`	22.61	21.25	21.15	135.00

As the table above shows, we eliminate around 10 clock cycles for each vector element with this simple transformation.

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 upper-case letters in a string to lower case. The procedure steps through the string, converting each upper-case 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 two. 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 hence 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 that attempted by even the most aggressive compilers. Programmers must do such transformations themselves.

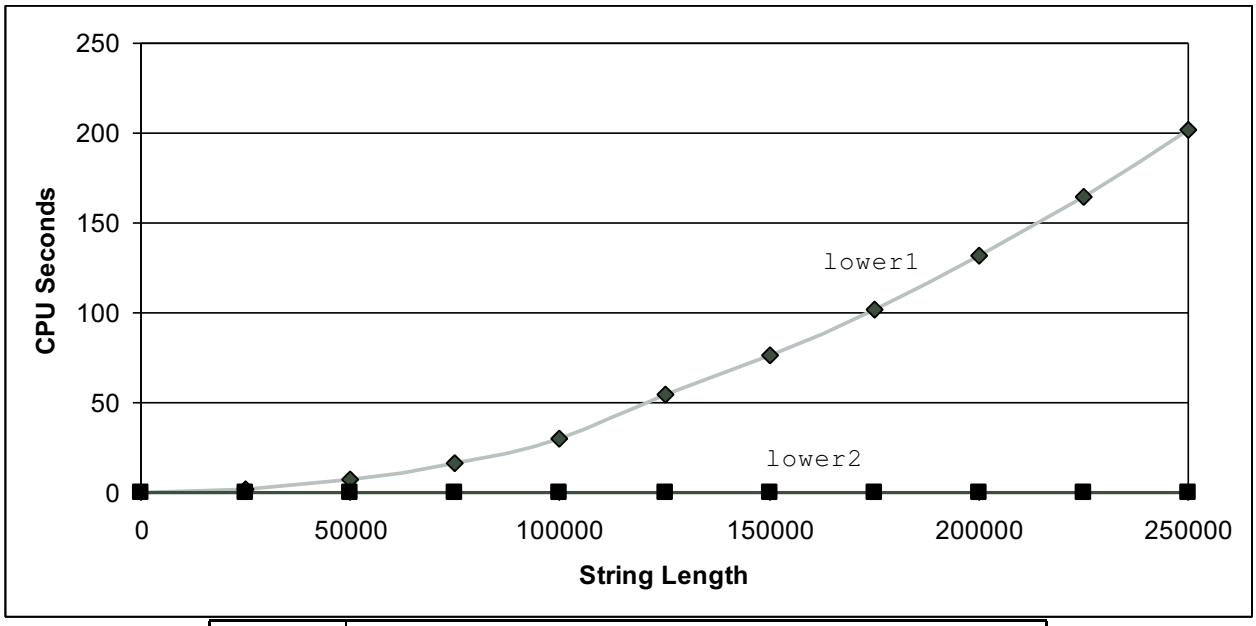
This example illustrates a common problem in writing programs, in which a seemingly trivial piece of code has a hidden asymptotic inefficiency. One would not expect a lower-case conversion routine to be a limiting factor in a program's performance. Typically, programs are tested and analyzed on small data sets, for which the performance of `lower1` is adequate. When the program is ultimately deployed, however, it is

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.

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 one 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 5.2:

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; }
```

Here are three code fragments that 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 table below indicating the number of times each of the four functions is called for each of these code fragments.

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.

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

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.

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 based on the following experimental results:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine2`	214	Move `vec_length`	20.66	21.25	21.15	135.00
`combine3`	219	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 where performance is a significant issue, one must often compromise modularity and abstraction for speed. It is wise to include documentation on the transformations applied, and the 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
```

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 1 on the next iteration will normally be the value that has just been written.

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     *dest = IDENT;
10    for (i = 0; i < length; i++) {
11        x = x OPER data[i];
12    }
13    *dest = x;
14 }
```

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.

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 following assembly code for the loop shows, the compiler can now use register `%eax` to hold the accumulated value. Comparing to the loop for `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:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine3`	219	Direct data access	6.00	9.00	8.00	117.00
`combine4`	221	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 above, `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 programs. At the assembly-code level, it appears as if instructions are executed one at a time, where each

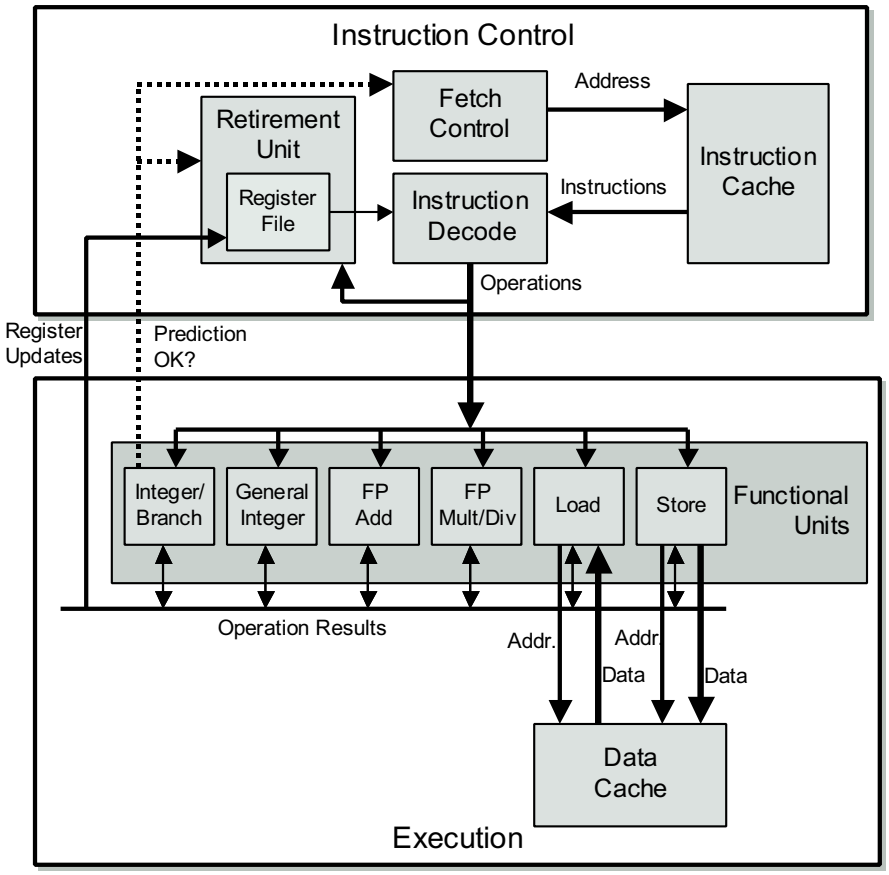


Figure 5.11: **Block Diagram of a Modern Processor.** The Instruction Control Unit is responsible for reading instructions from memory and generating a sequence of primitive operations. The Execution Unit then performs the operations and indicates whether the branches were correctly predicted.

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 [28], 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) is responsible for reading a sequence of instructions from memory and generating from these a set of primitive operations to perform on program data. The *Execution Unit* (EU) 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*, where they guess whether or not a branch will be taken, and they 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.

5.7.2 Functional Unit Performance

Figure 5.12 documents the performance of some of 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*, indicating the total number of cycles the functional unit requires to complete the operation; and the *issue time*, indicating the number of cycles between successive, independent operations. The latencies range from one cycle for basic integer operations; several cycles for loads, stores, integer multiplication, and the more common floating-point operations; and then 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 only be exploited 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 in 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 have also 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 one 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
```

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 taken. The jump operation

```
jl-taken 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`		

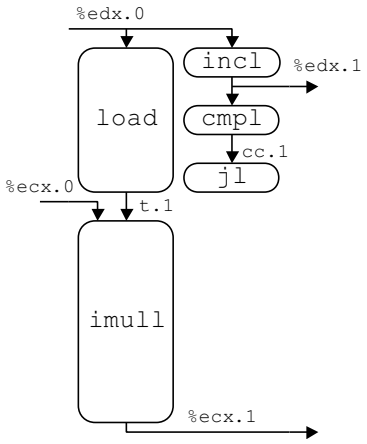


Figure 5.13: **Operations for First Iteration of Inner Loop of `combine4` for integer multiplication.** Memory reads are explicitly converted to loads. Register names are tagged with instance numbers.

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 as a *computation graph* where 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 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

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`		

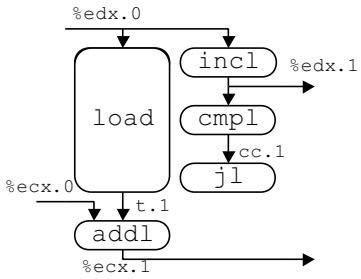


Figure 5.14: **Operations for First Iteration of Inner Loop of `combine4` for Integer Addition.** Compared to multiplication, the only change is that the addition operation requires only one cycle.

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, 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 4 cycles. Thus, the effective processing rate is one iteration every 4 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

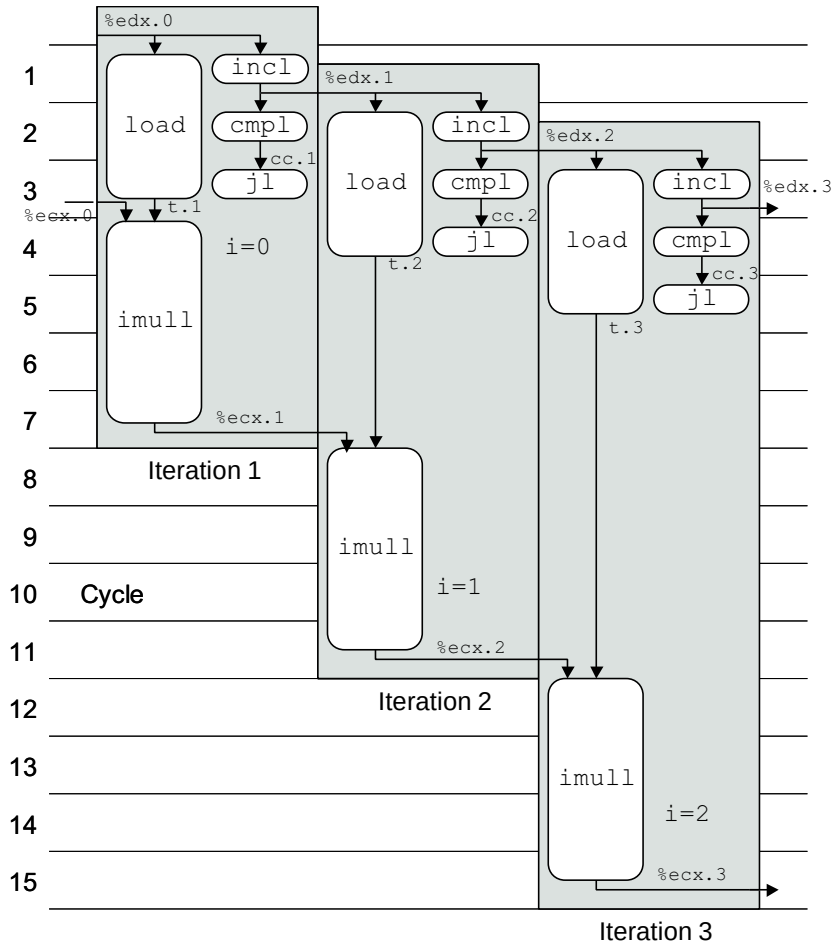


Figure 5.15: **Scheduling of Operations for Integer Multiplication with Unlimited Number of Execution Units.** The 4 cycle latency of the multiplier is the performance-limiting resource.

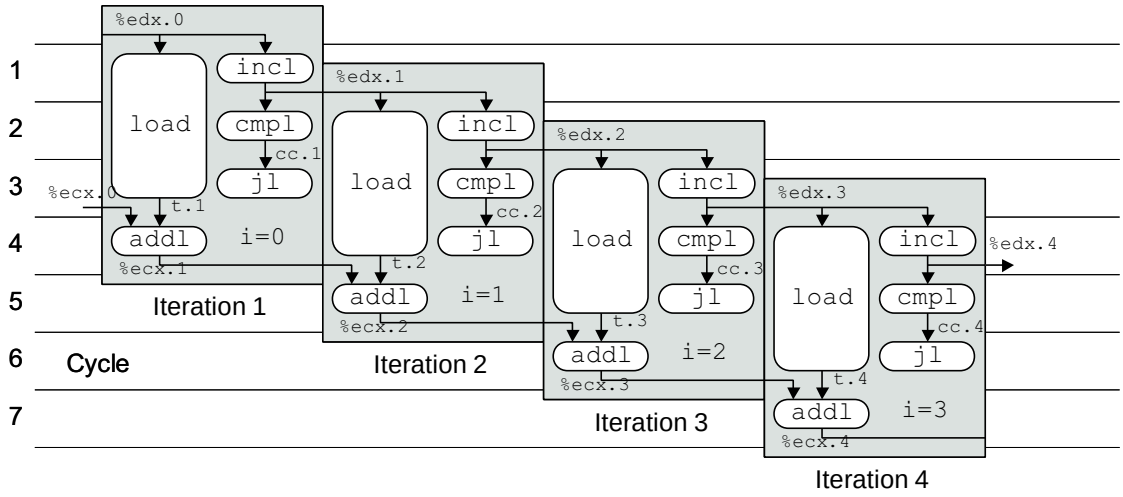


Figure 5.16: **Scheduling of Operations for Integer Addition with Unbounded Resource Constraints.** With unbounded resources the processor could achieve a CPE of 1.0.

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 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

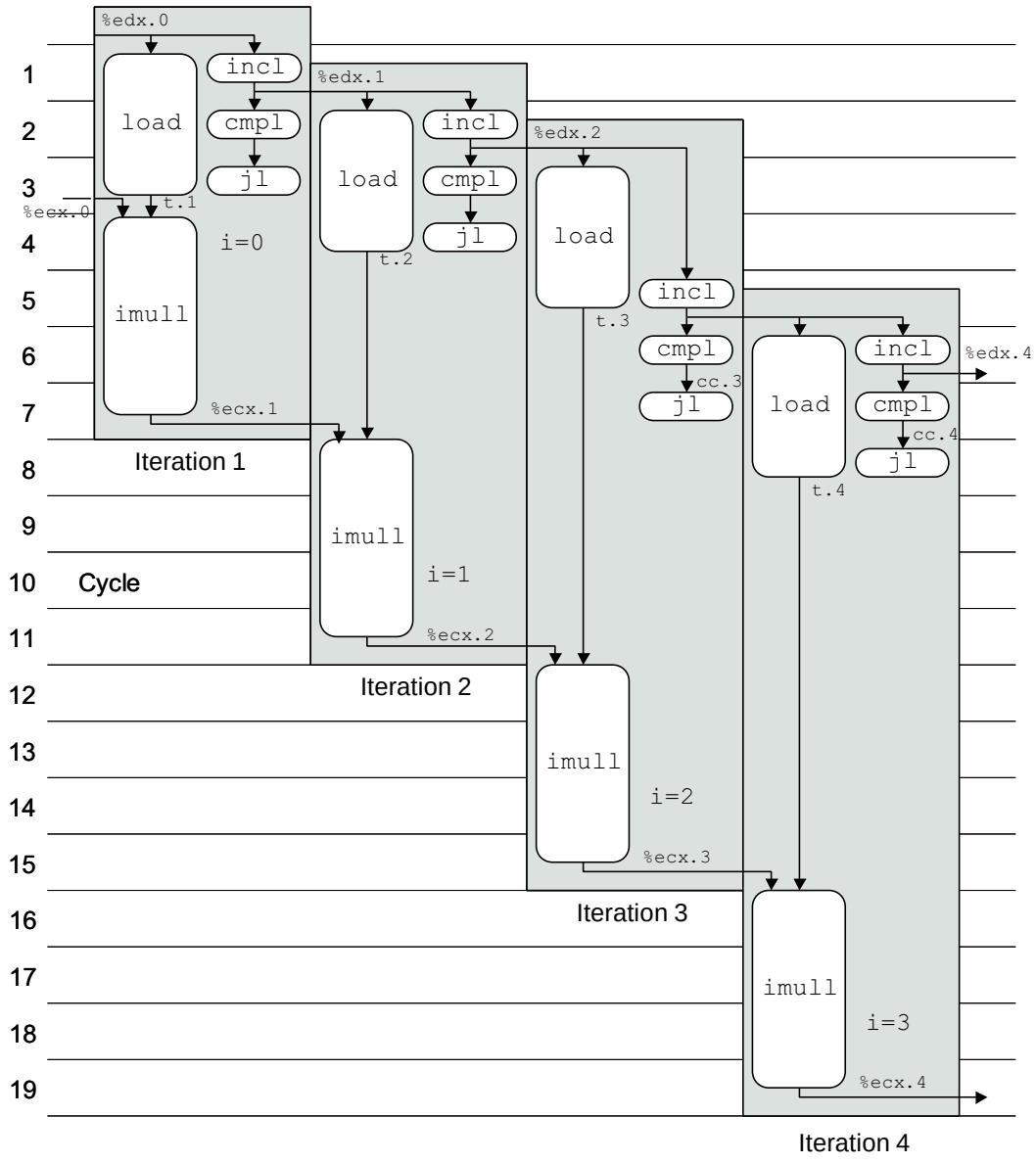


Figure 5.17: **Scheduling of Operations for Integer Multiplication with Actual Resource Constraints.** The multiplier latency remains the performance-limiting factor.

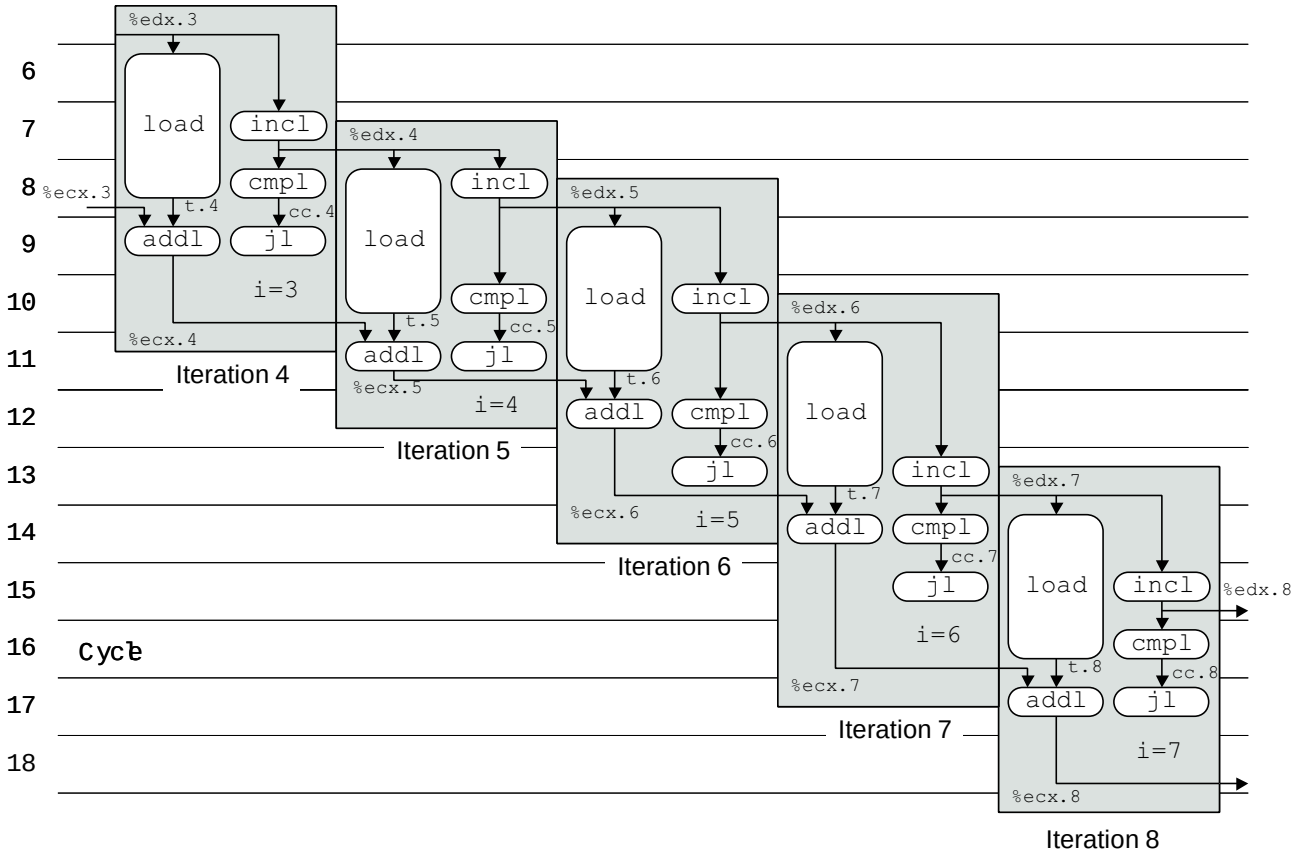


Figure 5.18: **Scheduling of Operations for Integer Addition with Actual Resource Constraints.** The limitation to two integer units constrains performance to a CPE of 2.0.

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

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine4`	221	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.

We can reduce overhead effects by performing more data operations in each iteration, via a technique known as *loop unrolling*. The idea is to access and combine multiple array elements within a single iteration. The resulting program requires fewer iterations, leading to reduced loop overhead.

Figure 5.19 shows a version of our combining code using three-way loop unrolling. The first loop steps through the array three elements at a time. That is, the loop index `i` is incremented by three on each iteration, and the combining operation is applied to array elements i, $i + 1$, and $i + 2$ in a single iteration.

code/opt/combine.c

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

code/opt/combine.c

Figure 5.19: **Unrolling Loop by 3.** Loop unrolling can reduce the effect of loop overhead.

Execution Unit Operations		
load (%eax, %edx.0, 4)	→	t.1a
addl t.1a, %ecx.0c	→	%ecx.1a
load 4(%eax, %edx.0, 4)	→	t.1b
addl t.1b, %ecx.1a	→	%ecx.1b
load 8(%eax, %edx.0, 4)	→	t.1c
addl t.1c, %ecx.1b	→	%ecx.1c
addl %edx.0, 3	→	%edx.1
cmpl %esi, %edx.1	→	cc.1
jl-taken cc.1		

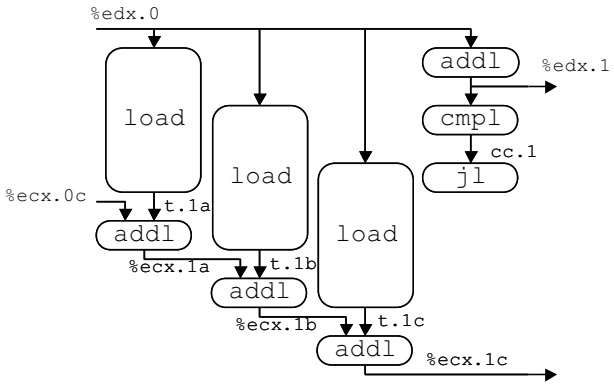


Figure 5.20: **Operations for First Iteration of Inner Loop of Three-Way Unrolled Integer Addition.** With this degree of loop unrolling we can combine three array elements using six integer/branch operations.

In general, the vector length will not be a multiple of 3. 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 $n-2$. We are then assured that the loop will only be executed when the loop index i satisfies $i < n-2$, and hence the maximum array index $i+2$ will satisfy $i+2 < (n-2)+2 = n$. In general, if the loop is unrolled by k, we set the upper limit to be $n-k+1$. The maximum loop index $i+k-1$ will then be less than n. In addition to this, we add a 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 2 times.

To better understand the performance of code with loop unrolling, let us look at the assembly code for the inner loop and its translation into operations.

Assembly Instructions	Execution Unit Operations		
.L49:			
addl (%eax,%edx,4),%ecx	load (%eax, %edx.0, 4)	→	t.1a
	addl t.1a, %ecx.0c	→	%ecx.1a
addl 4(%eax,%edx,4),%ecx	load 4(%eax, %edx.0, 4)	→	t.1b
	addl t.1b, %ecx.1a	→	%ecx.1b
addl 8(%eax,%edx,4),%ecx	load 8(%eax, %edx.0, 4)	→	t.1c
	addl t.1c, %ecx.1b	→	%ecx.1c
addl %edx,3	addl %edx.0, 3	→	%edx.1
cmpl %esi,%edx	cmpl %esi, %edx.1	→	cc.1
jl .L49	jl-taken cc.1		

As mentioned earlier, loop unrolling by itself will only help the performance of the code for the case of integer sum, since our other cases are limited by the latency of the functional units. For integer sum, three-way unrolling allows us to combine three elements with six integer/branch operations, as shown in Figure 5.20. With two functional units for these operations, we could potentially achieve a CPE of 1.0. Figure 5.21

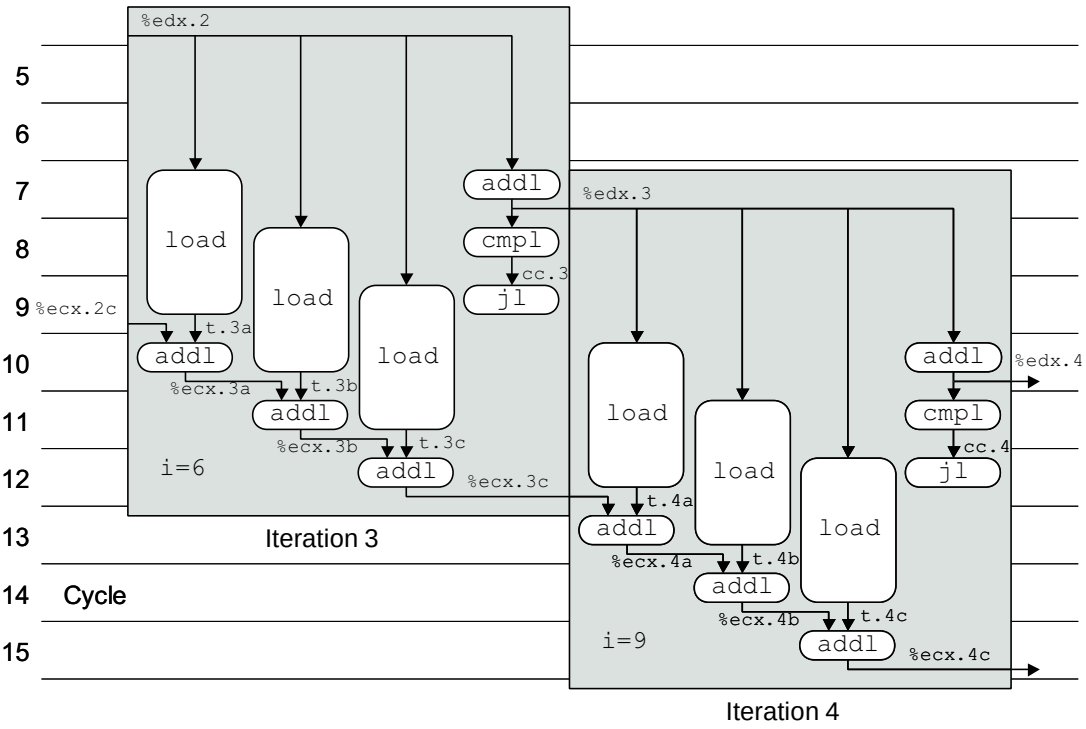


Figure 5.21: **Scheduling of Operations for Iterations 2 to 3 with Three-Way Unrolled Integer Sum with Bounded Resource Constraints.** In principle, the procedure can achieve a CPE of 1.0. The measured CPE, however, is 1.33.

shows that once we reach iteration 3 (`i` $= 6$), the operations would follow a regular pattern. The operations of iteration 4 (`i` $= 9$) have the same timings, but shifted by three cycles. This would indeed yield a CPE of 1.0.

Our measurement for this function shows a CPE of 1.33, that is, we require four cycles per iteration. Evidently some resource constraint we did not account for in our analysis delays the computation by one additional cycle per iteration. Nonetheless, this performance represents an improvement over the code without loop unrolling.

Measuring the performance for different degrees of unrolling yields the following values for the CPE

Vector Length	Degree of Unrolling					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06

As these measurements show, loop unrolling can reduce the CPE. With the loop unrolled by a factor of two, each iteration of the main loop requires three clock cycles, giving a CPE of $3/2 = 1.5$. As we increase the degree of unrolling, we generally get better performance, nearing the theoretical CPE limit of 1.0. It is interesting to note that the improvement is not monotonic—unrolling by three gives better performance than unrolling by four. Evidently the scheduling of operations on the execution units is less efficient for the latter case.

Our CPE measurements do not account for overhead factors such as the cost of the procedure call and of setting up the loop. With loop unrolling, we introduce a new source of overhead—the need to finish any remaining elements when the vector length is not divisible by the degree of unrolling. To investigate the impact of overhead, we measure the *net CPE* for different vector lengths. The net CPE is computed as the total number of cycles required by the procedure divided by the number of elements. For the different degrees of unrolling, and for two different vector lengths we obtain the following:

Vector Length	Degree of Unrolling					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06
31 Net CPE	4.02	3.57	3.39	3.84	3.91	3.66
1024 Net CPE	2.06	1.56	1.40	1.56	1.31	1.12

The distinction between CPE and net CPE is minimal for long vectors, as seen with the measurements for length 1024, but the impact is significant for short vectors, as seen with the measurements for length 31. Our measurements of the net CPE for a vector of length 31 demonstrate one drawback of loop unrolling. Even with no unrolling, the net CPE of 4.02 is considerably higher than the 2.06 measured for long vectors. The overhead of starting and completing the loop becomes far more significant when the loop is executed a smaller number of times. In addition, the benefit of loop unrolling is less significant. Our unrolled code must start and stop two loops, and it must complete the final elements one at a time. The overhead decreases with increased loop unrolling, while the number of operations performed in the final loop increases. With a vector length of 1024, performance generally improves as the degree of unrolling increases. With a vector length of 31, the best performance is achieved by unrolling the loop by only a factor of three.

A second drawback of loop unrolling is that it increases the amount of object code generated. The object code for `combine4` requires 63 bytes, whereas the object code with the loop unrolled by a factor of 16

requires 142 bytes. In this case, that seems like a small price to pay for code that runs nearly twice as fast. In other cases, however, the optimum position in this time-space tradeoff is not so clear.

5.9 Converting to Pointer Code

Before proceeding further, let us attempt one more transformation that can sometimes improve program performance, at the expense of program readability. One of the unique features of C is the ability to create and reference pointers to arbitrary program objects. Pointer arithmetic, in fact, has a close connection to array referencing. The combination of pointer arithmetic and referencing given by the expression `*(a+i)` is exactly equivalent to the array reference `a[i]`. At times, we can improve the performance of a program by using pointers rather than arrays.

Figure 5.22 shows an example of converting the procedures `combine4` and `combine5` to pointer code, giving procedures `combine4p` and `combine5p`, respectively. Instead of keeping pointer `data` fixed at the beginning of the vector, we move it with each iteration. The vector elements are then referenced by a fixed offset (between 0 and 2) of `data`. Most significantly, we can eliminate the iteration variable `i` from the procedure. To detect when the loop should terminate, we compute a pointer `dend` to be an upper bound on pointer `data`.

Comparing the performance of these procedures to their array counterparts yields mixed results:

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine4`	221	Accumulate in temporary	2.00	4.00	3.00	5.00
`combine4p`	241	Pointer version	**3.00**	4.00	3.00	5.00
`combine5`	236	Unroll loop $\times 3$	1.33	4.00	3.00	5.00
`combine5p`	241	Pointer version	1.33	4.00	3.00	5.00
`combine5x4`		Unroll loop $\times 4$	1.50	4.00	3.00	5.00
`combine5px4`		Pointer version	**1.25**	4.00	3.00	5.00

For most of the cases, the array and pointer versions have the exact same performance. With pointer code, the CPE for integer sum with no unrolling actually gets worse by one cycle. This result is somewhat surprising, since the inner loops for the pointer and array versions are very similar, as shown in Figure 5.23. It is hard to imagine why the pointer code requires an additional clock cycle per iteration. Just as mysteriously, versions of the procedures with four-way loop unrolling yield a one-cycle-per-iteration improvement with pointer code, giving a CPE of 1.25 (five cycles per iteration) rather then 1.5 (six cycles per iteration).

In our experience, the relative performance of pointer versus array code depends on the machine, the compiler, and even the particular procedure. We have seen compilers that apply very advanced optimizations to array code but only minimal optimizations to pointer code. For the sake of readability, array code is generally preferable.

Practice Problem 5.3:

At times, GCC does its own version of converting array code to pointer code. For example, with integer data and addition as the combining operation, it generates the following code for the inner loop of a variant of `combine5` that uses eight-way loop unrolling:

code/opt/combine.c

```
1 /* Accumulate in local variable, pointer version */
2 void combine4p(vec_ptr v, data_t *dest)
3 {
4     int length = vec_length(v);
5     data_t *data = get_vec_start(v);
6     data_t *dend = data+length;
7     data_t x = IDENT;
8
9     for (; data < dend; data++)
10        x = x OPER *data;
11    *dest = x;
12 }
```

code/opt/combine.c

(a) Pointer version of `combine4`.

code/opt/combine.c

```
1 /* Unroll loop by 3, pointer version */
2 void combine5p(vec_ptr v, data_t *dest)
3 {
4     data_t *data = get_vec_start(v);
5     data_t *dend = data+vec_length(v);
6     data_t *dlimit = dend-2;
7     data_t x = IDENT;
8
9     /* Combine 3 elements at a time */
10    for (; data < dlimit; data += 3) {
11        x = x OPER data[0] OPER data[1] OPER data[2];
12    }
13
14    /* Finish any remaining elements */
15    for (; data < dend; data++) {
16        x = x OPER data[0];
17    }
18    *dest = x;
19 }
```

code/opt/combine.c

(b) Pointer version of `combine5`

Figure 5.22: **Converting Array Code to Pointer Code.** In some cases, this can lead to improved performance.

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

(a) Array code

```
    combine4p: type=INT, OPER = '+'
    data in %eax, x in %ecx, dend in %edx
1 .L30:                          loop:
2   addl (%eax),%ecx             Add data[0] to x
3   addl $4,%eax                 data++
4   cmpl %edx,%eax               Compare data:dend
5   jb .L30                      If <, goto loop
```

(b) Pointer code

Figure 5.23: **Pointer Code Performance Anomaly.** Although the two programs are very similar in structure, the array code requires two cycles per iteration, while the pointer code requires three.

```
 1 .L6:
 2   addl (%eax),%edx
 3   addl 4(%eax),%edx
 4   addl 8(%eax),%edx
 5   addl 12(%eax),%edx
 6   addl 16(%eax),%edx
 7   addl 20(%eax),%edx
 8   addl 24(%eax),%edx
 9   addl 28(%eax),%edx
10   addl $32,%eax
11   addl $8,%ecx
12   cmpl %esi,%ecx
13   jl .L6
```

Observe how register `%eax` is being incremented by 32 on each iteration.

Write C code for a procedure `combine5px8` that shows how pointers, loop variables, and termination conditions are being computed by this code. Show the general form with arbitrary data and combining operation in the style of Figure 5.19. Describe how it differs from our handwritten pointer code (Figure 5.22).

code/opt/combine.c

```
1 /* Unroll loop by 2, 2-way parallelism */
2 void combine6(vec_ptr v, data_t *dest)
3 {
4     int length = vec_length(v);
5     int limit = length-1;
6     data_t *data = get_vec_start(v);
7     data_t x0 = IDENT;
8     data_t x1 = IDENT;
9     int i;
10
11    /* Combine 2 elements at a time */
12    for (i = 0; i < limit; i+=2) {
13        x0 = x0 OPER data[i];
14        x1 = x1 OPER data[i+1];
15    }
16
17    /* Finish any remaining elements */
18    for (; i < length; i++) {
19        x0 = x0 OPER data[i];
20    }
21    *dest = x0 OPER x1;
22 }
```

code/opt/combine.c

Figure 5.24: **Unrolling Loop by 2 and Using Two-Way Parallelism.** This approach makes use of the pipelining capability of the functional units.

5.10 Enhancing Parallelism

At this point, our programs are limited by the latency of the functional units. As the third column in Figure 5.12 shows, however, several functional units of the processor are *pipelined*, meaning that they can start on a new operation before the previous one is completed. Our code cannot take advantage of this capability, even with loop unrolling, since we are accumulating the value as a single variable `x`. We cannot compute a new value of `x` until the preceding computation has completed. As a result, the processor will *stall*, waiting to begin a new operation until the current one has completed. This limitation shows clearly in Figures 5.15 and 5.17. Even with unbounded processor resources, the multiplier can only produce a new result every four clock cycles. Similar limitations occur with floating-point addition (three cycles) and multiplication (five cycles).

5.10.1 Loop Splitting

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

Execution Unit Operations		
`load (%eax, %edx.0, 4)`	→	`t.1a`
`imull t.1a, %ecx.0`	→	`%ecx.1`
`load 4(%eax, %edx.0, 4)`	→	`t.1b`
`imull t.1b, %ebx.0`	→	`%ebx.1`
`addl $2, %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

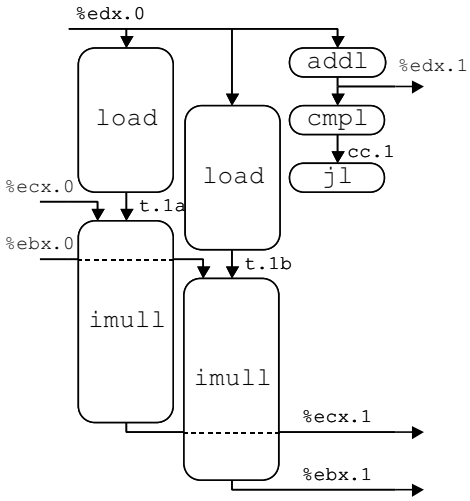


Figure 5.25: **Operations for First Iteration of Inner Loop of Two-Way Unrolled, Two-Way Parallel Integer Multiplication.** The two multiplication operations are logically independent.

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:

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

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

Figure 5.24 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 index in variable `x0`, and elements with odd index in variable `x1`. 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 `x0` and `x1` to compute the final result.

To see how this code yields improved performance, let us consider the translation of the loop into operations for the case of integer multiplication:

Assembly Instructions	Execution Unit Operations
`.L151:`	
`imull (%eax,%edx,4),%ecx`	`load (%eax, %edx.0, 4)` → `t.1a`
	`imull t.1a, %ecx.0` → `%ecx.1`
`imull 4(%eax,%edx,4),%ebx`	`load 4(%eax, %edx.0, 4)` → `t.1b`
	`imull t.1b, %ebx.0` → `%ebx.1`
`addl $2,%edx`	`addl $2, %edx.0` → `%edx.1`
`cmpl %esi,%edx`	`cmpl %esi, %edx.1` → `cc.1`
`jl .L151`	`jl-taken cc.1`

Figure 5.25 shows a graphical representation of these operations for the first iteration (`i` $= 0$). As this diagram illustrates, the two multiplications in the loop are independent of each other. One has register `%ecx` as its source and destination (corresponding to program variable `x0`), while the other has register `%ebx` as its source and destination (corresponding to program variable `x1`). The second multiplication can start just one cycle after the first. This makes use of the pipelining capabilities of both the load unit and the integer multiplier.

Figure 5.26 shows a graphical representation of the first three iterations (`i` $= 0$, 2, and 4) for integer multiplication. For each iteration, the two multiplications must wait until the results from the previous iteration have been computed. Still, the machine can generate two results every four clock cycles, giving a theoretical CPE of 2.0. In this figure we do not take into account the limited set of integer functional units, but this does not prove to be a limitation for this particular procedure.

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

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
		Unroll $\times 2$	1.50	4.00	3.00	5.00
`combine6`	243	Unroll $\times 2$, Parallelism $\times 2$	1.50	2.00	2.00	2.50

For integer sum, parallelism does not help, as the latency of integer addition is only one clock cycle. For integer and floating-point product, however, we reduce the CPE by a factor of two. We are essentially doubling the use of the functional units. For floating-point sum, some other resource constraint is limiting our CPE to 2.0, rather than the theoretical value of 1.5.

We have seen earlier 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. This is referred to as *iteration splitting* in the optimizing compiler literature. Many compilers do loop unrolling automatically, but relatively few do iteration splitting.

On the other hand, we have seen that floating-point multiplication and addition are not associative. Thus, `combine5` and `combine6` could potentially produce different results due to rounding or overflow. Imagine, for example, a case where all the elements with even indices were numbers with very large absolute value, while those with odd indices were very close to 0.0. Then product PE_n might overflow, or PO_n might underflow, even though the final product P_n does not. In most real-life applications, however, such

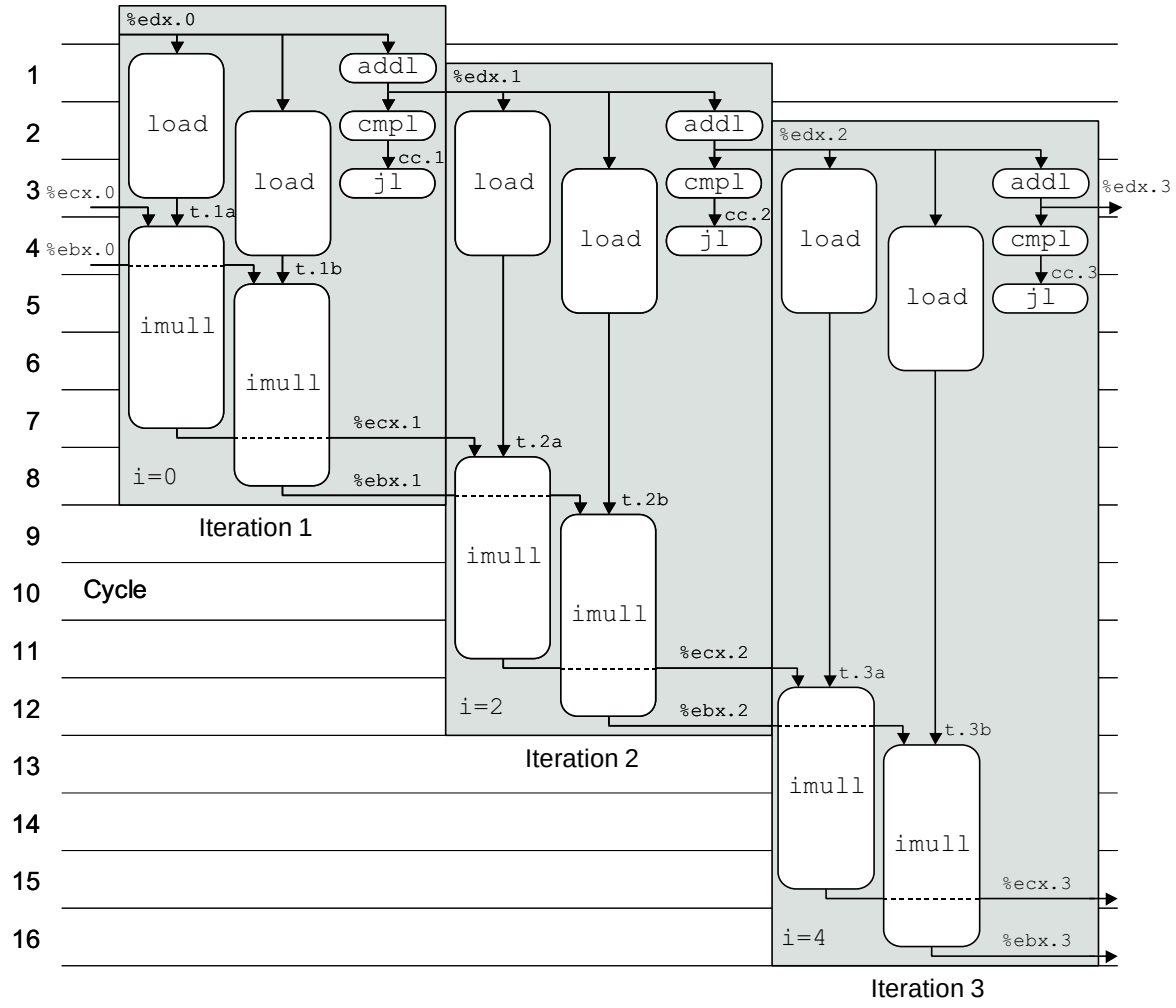


Figure 5.26: **Scheduling of Operations for Two-Way Unrolled, Two-Way Parallel Integer Multiplication with Unlimited Resources.** The multiplier can now generate two values every 4 cycles.

patterns are unlikely. Since most physical phenomena are continous, 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 above. It is unlikely that summing the elements in strict order gives fundamentally better accuracy than does summing two groups independently and then adding those sums together. For most applications, achieving a performance gain of 2X 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.

Just as we can unroll loops by an arbitrary factor k, we can also increase the parallelism to any factor p such that k is divisible by p. The following are some results for different degrees of unrolling and parallelism:

Method	Integer		Floating Point	
	+	*	+	*
Unroll $\times 2$	1.50	4.00	3.00	5.00
Unroll $\times 2$, Parallelism $\times 2$	1.50	2.00	2.00	2.50
Unroll $\times 4$	1.50	4.00	3.00	5.00
Unroll $\times 4$, Parallelism $\times 2$	1.50	2.00	1.50	2.50
Unroll $\times 8$	1.25	4.00	3.00	5.00
Unroll $\times 8$, Parallelism $\times 2$	1.25	2.00	1.50	2.50
Unroll $\times 8$, Parallelism $\times 4$	1.25	1.25	1.61	2.00
Unroll $\times 8$, Parallelism $\times 8$	1.75	1.87	1.87	2.07
Unroll $\times 9$, Parallelism $\times 3$	1.22	1.33	1.66	2.00

As this table shows, increasing the degree of loop unrolling and the degree of parallelism helps program performance up to some point, but it yields diminishing improvement or even worse performance when taken to an extreme. In the next section, we will describe two reasons for this phenomenon.

5.10.2 Register Spilling

The benefits of loop parallelism are limited by the ability to express the computation in assembly code. In particular, the IA32 instruction set only has a small number of registers to hold the values being accumulated. If we have 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 on the stack. Once this happens, the performance drops dramatically. This occurs for our benchmarks when we attempt to have $p = 8$. Our measurements show the performance for this case is worse than that for $p = 4$.

For the case of the integer data type, there are only eight total integer registers available. Two of these (`%ebp` and `%esp`) point to regions of the stack. With the pointer version of the code, one of the remaining six holds the pointer `data`, and one holds the stopping position `dend`. This leaves only four integer registers for accumulating values. With the array version of the code, we require three registers to hold the loop index `i`, the stopping index `limit`, and the array address `data`. This leaves only three registers for accumulating values. For the floating-point data type, we need two of eight registers to hold intermediate values, leaving six for accumulating values. Thus, we could have a maximum parallelism of six before register spilling occurs.

This limitation to eight integer and eight floating-point registers is an unfortunate artifact of the IA32 instruc-

tion set. The renaming scheme described previously eliminates the direct correspondence between register names and the actual location of the register data. In a modern processor, register names serve simply to identify the program values being passed between the functional units. IA32 provides only a small number of such identifiers, constraining the amount of parallelism that can be expressed in programs.

The occurrence of spilling can be seen by examining the assembly code. For example, within the first loop for the code with eight-way parallelism we see the following instruction sequence:

```
     type=INT, OPER = '*'
     x6 in -12(%ebp), data+i in %eax
1    movl -12(%ebp),%edi       Get x6 from stack
2    imull 24(%eax),%edi       Multiply by data[i+6]
3    movl %edi,-12(%ebp)       Put x6 back
```

In this code, a stack location is being used to hold `x6`, one of the eight local variables used to accumulate sums. The code loads it into a register, multiplies it by one of the data elements, and stores it back to the same stack location. As a general rule, any time a compiled program shows evidence of register spilling within some heavily used inner loop, it might be preferable to rewrite the code so that fewer temporary values are required. These include explicitly declared local variables as well as intermediate results being saved to avoid recomputation.

Practice Problem 5.4:

The following shows the code generated from a variant of `combine6` that uses eight-way loop unrolling and four-way parallelism.

```
 1 .L152:
 2   addl (%eax),%ecx
 3   addl 4(%eax),%esi
 4   addl 8(%eax),%edi
 5   addl 12(%eax),%ebx
 6   addl 16(%eax),%ecx
 7   addl 20(%eax),%esi
 8   addl 24(%eax),%edi
 9   addl 28(%eax),%ebx
10   addl $32,%eax
11   addl $8,%edx
12   cmpl -8(%ebp),%edx
13   jl .L152
```

A. What program variable has being spilled onto the stack?

B. At what location on the stack?

C. Why is this a good choice of which value to spill?

With floating-point data, we want to keep all of the local variables in the floating-point register stack. We also need to keep the top of stack available for loading data from memory. This limits us to a degree of parallelism less than or equal to 7.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	213	Abstract unoptimized	42.06	41.86	41.44	160.00
`combine1`	213	Abstract `-O2`	31.25	33.25	31.25	143.00
`combine2`	214	Move `vec_length`	20.66	21.25	21.15	135.00
`combine3`	219	Direct data access	6.00	9.00	8.00	117.00
`combine4`	221	Accumulate in temporary	2.00	4.00	3.00	5.00
`combine5`	236	Unroll $\times 4$	1.50	4.00	3.00	5.00
		Unroll $\times 16$	**1.06**	4.00	3.00	5.00
`combine6`	243	Unroll $\times 2$, Parallelism $\times 2$	1.50	2.00	2.00	2.50
		Unroll $\times 4$, Parallelism $\times 2$	1.50	2.00	1.50	2.50
		Unroll $\times 8$, Parallelism $\times 4$	1.25	**1.25**	**1.50**	**2.00**
Worst:Best			39.7	33.5	27.6	80.0

Figure 5.27: **Comparative Result for All Combining Routines.** The best performing version is shown in bold face.

5.10.3 Limits to Parallelism

For our benchmarks, the main performance limitations are due to the capabilities of the functional units. As Figure 5.12 shows, the integer multiplier and the floating-point adder can only initiate a new operation every clock cycle. This, plus a similar limitation on the load unit limits these cases to a CPE of 1.0. The floating-point multiplier can only initiate a new operation every two clock cycles. This limits this case to a CPE of 2.0. Integer sum is limited to a CPE of 1.0, due to the limitations of the load unit. This leads to the following comparison between the achieved performance versus the theoretical limits:

Method	Integer		Floating Point	
	+	*	+	*
Achieved	1.06	1.25	1.50	2.00
Theoretical Limit	1.00	1.00	1.00	2.00

In this table, we have chosen the combination of unrolling and parallelism that achieves the best performance for each case. We have been able to get close to the theoretical limit for integer sum and product and for floating-point product. Some machine-dependent factor limits the achieved CPE for floating-point multiplication to 1.50 rather than the theoretical limit of 1.0.

5.11 Putting it Together: Summary of Results for Optimizing Combining Code

We have now considered six versions of the combining code, some of which had multiple variants. Let us pause to take a look at the overall effect of this effort, and how our code would do on a different machine.

Figure 5.27 shows the measured performance for all of our routines plus several other variants. As can

be seen, we achieve maximum performance for the integer sum by simply unrolling the loop many times, whereas we achieve maximum performance for the other operations by introducing some, but not too much, parallelism. The overall performance gain of 27.6X and better from our original code is quite impressive.

5.11.1 Floating-Point Performance Anomaly

One of the most striking features of Figure 5.27 is the dramatic drop in the cycle time for floating-point multiplication when we go from `combine3`, where the product is accumulated in memory, to `combine4` where the product is accumulated in a floating-point register. By making this small change, the code suddenly runs 23.4 times faster. When an unexpected result such as this one arises, it is important to hypothesize what could cause this behavior and then devise a series of tests to evaluate this hypothesis.

Examining the table, it appears that something strange is happening for the case of floating-point multiplication when we accumulate the results in memory. The performance is far worse than for floating-point addition or integer multiplication, even though the number of cycles for the functional units are comparable. On an IA32 processor, all floating-point operations are performed in extended 80-bit) precision, and the floating-point registers store values in this format. Only when the value in a register is written to memory is it converted to 32-bit (`float`) or 64-bit (`double`) format.

Examining the data used for our measurements, the source of the problem becomes clear. The measurements were performed on a vector of length 1024 having element i equal to $i + 1$. Hence, we are attempting to compute $1024!$, which is approximately 5.4×10^{2639}. Such a large number can be represented in the extended-precision floating-point format (it can represent numbers up to around 10^{4932}), but it far exceeds what can be represented as a single precision (up to around 10^{38}) or double precision (up to around 10^{308}). The single precision case overflows when we reach $i = 34$, while the double precision case overflows when we reach $i = 171$. Once we reach this point, every execution of the statement

```
*dest = *dest OPER val;
```

in the inner loop of `combine3` requires reading the value $+\infty$, from `dest`, multiplying this by `val` to get $+\infty$ and then storing this back at `dest`. Evidently, some part of this computation requires much longer than the normal five clock cycles required by floating-point multiplication. In fact, running measurements on this operation we find it takes between 110 and 120 cycles to multiply a number by infinity. Most likely, the hardware detected this as a special case and issued a *trap* that caused a software routine to perform the actual computation. The CPU designers felt such an occurrence would be sufficiently rare that they did not need to deal with it as part of the hardware design. Similar behavior could happen with underflow.

When we run the benchmarks on data for which every vector element equals 1.0, `combine3` achieves a CPE of 10.00 cycles for both double and single precision. This is much more in line with the times measured for the other data types and operations, and comparable to the time for `combine4`.

This example illustrates one of the challenges of evaluating program performance. Measurements can be strongly affected by characteristics of the data and operating conditions that initially seem insignificant.

Function	Page	Method	Integer		Floating Point	
			+	*	+	*
`combine1`	213	Abstract unoptimized	40.14	47.14	52.07	53.71
`combine1`	213	Abstract `-O2`	25.08	36.05	37.37	32.02
`combine2`	214	Move `vec_length`	19.19	32.18	28.73	32.73
`combine3`	219	Direct data access	6.26	12.52	13.26	13.01
`combine4`	221	Accumulate in temporary	1.76	9.01	8.01	8.01
`combine5`	236	Unroll $\times 4$	1.51	9.01	6.32	6.32
		Unroll $\times 16$	1.25	9.01	6.33	6.22
`combine6`	243	Unroll $\times 4$, Parallelism $\times 2$	1.19	4.69	4.44	4.45
		Unroll $\times 8$, Parallelism $\times 4$	1.15	**4.12**	2.34	**2.01**
		Unroll $\times 8$, Parallelism $\times 8$	**1.11**	4.24	**2.36**	2.08
Worst:Best			36.2	11.4	22.3	26.7

Figure 5.28: **Comparative Result for All Combining Routines Running on a Compaq Alpha 21164 Processor.** The same general optimization techniques are useful on this machine as well.

5.11.2 Changing Platforms

Although we presented our optimization strategies in the context of a specific machine and compiler, the general principles also apply to other machine and compiler combinations. Of course, the optimal strategy may be very machine dependent. As an example, Figure 5.28 shows performance results for a Compaq Alpha 21164 processor for conditions comparable to those for a Pentium III shown in Figure 5.27. These measurements were taken for code generated by the Compaq C compiler, which applies more advanced optimizations than GCC. Observe how the cycle times generally decline as we move down the table, just as they did for the other machine. We see that we can effectively exploit a higher (eight-way) degree of parallelism, because the Alpha has 32 integer and 32 floating-point registers. As this example illustrates, the general principles of program optimization apply to a variety of different machines, even if the particular combination of features leading to optimum performance depend on the specific machine.

5.12 Branch Prediction and Misprediction Penalties

As we have mentioned, 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, however, 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

code/opt/absval.c

```
1 int absval(int val)
2 {
3     return (val<0) ? -val : val;
4 }
```

code/opt/absval.c

(a) C code.

```
1  absval:
2    pushl %ebp
3    movl %esp,%ebp
4    movl 8(%ebp),%eax    Get val
5    testl %eax,%eax      Test it
6    jge .L3              If >0, goto end
7    negl %eax            Else, negate it
8  .L3:                   end:
9    movl %ebp,%esp
10   popl %ebp
11   ret
```

(b) Assembly code.

Figure 5.29: **Absolute Value Code** We use this to measure the cost of branch misprediction.

until the actual outcome has been determined. If the prediction is correct, the processor simply "commits" 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. A significant *branch penalty* is incurred in doing this, because the instruction pipeline must be refilled before useful results are generated.

Once upon a time, the technology required to support speculative execution was considered too costly and exotic for all but the most advanced supercomputers. Since around 1998, integrated circuit technology has made it possible to put so much circuitry on one chip that some can be dedicated to supporting branch prediction and speculative execution. At this point, almost every processor in a desktop or server machine supports speculative execution.

In optimizing our combining procedure, we did not observe any performance limitation imposed by the loop structure. That is, it appeared that the only limiting factor to performance was due to the functional units. For this procedure, the processor was generally able to predict the direction of the branch at the end of the loop. In fact, if it predicted the branch will always be taken, the processor would be correct on all but the final iteration.

Many schemes have been devised for predicting branches, and many studies have been made on their performance. A common heuristic is to predict that any branch to a lower address will be taken, while any branch to a higher address will not be taken. Branches to lower addresses are used to close loops, and since loops are usually executed many times, predicting these branches as being taken is generally a good idea. Forward branches, on the other hand, are used for conditional computation. Experiments have shown that the backward-taken, forward-not-taken heuristic is correct around 65% of the time. Predicting all branches as being taken, on the other other hand, has a success rate of only around 60%. Far more sophisticated strategies have been devised, requiring greater amounts of hardware. For example, the Intel Pentium II and III processors use a branch prediction strategy that is claimed to be correct between 90% and 95% of the time [29].

We can run experiments to test the branch predication capability of a processor and the cost of a misprediction. We use the absolute value routine shown in Figure 5.29 as our test case. This figure also shows the compiled form. For nonnegative arguments, the branch will be taken to skip over the negation instruction.

We time this function computing the absolute value of every element in an array, with the array consisting of various patterns of of $+1$s and -1s. For regular patterns (e.g., all $+1$s, all -1s, or alternating $+1$ and -1s), we find the function requires between 13.01 and 13.41 cycles. We use this as our estimate of the performance with perfect branch condition. On an array set to random patterns of $+1$s and -1s, we find that the function requires 20.32 cycles. One principle of random processes is that no matter what strategy one uses to guess a sequence of values, if the underlying process is truly random, then we will be right only 50% of the time. For example, no matter what strategy one uses to guess the outcome of a coin toss, as long as the coin toss is fair, our probability of success is only 0.5. Thus, we can see that a mispredicted branch with this processor incurs a penalty of around 14 clock cycles, since a misprediction rate of 50% causes the function to run an average of 7 cycles slower. This means that calls to `absval` require between 13 and 27 cycles depending on the success of the branch predictor.

This penalty of 14 cycles is quite large. For example, if our prediction accuracy were only 65%, then the processor would waste, on average, $14 \times 0.35 = 4.9$ cycles for every branch instruction. Even with the 90 to 95% prediction accuracy claimed for the Pentium II and III, around one cycle is wasted for every branch due to mispredictions. Studies of actual programs show that branches constitute around 14 to 16% of all executed instructions in typical "integer" programs (i.e., those that do not process numeric data), and around 3 to 12% of all executed instructions in typical numeric programs[31, Sect. 3.5]. Thus, any wasted time due to inefficient branch handling can have a significant effect on processor performance.

Many data dependent branches are not at all predictable. For example, there is no basis for guessing whether an argument to our absolute value routine will be positive or negative. To improve performance on code involving conditional evaluation, many processor designs have been extended to include *conditional move* instructions. These instructions allow some forms of conditionals to be implemented without any branch instructions.

With the IA32 instruction set, a number of different `cmov` instructions were added starting with the PentiumPro. These are supported by all recent Intel and Intel-compatible processors. These instructions perform an operation similar to the C code:

```
if (COND)
    x = y;
```

where `y` is the source operand and `x` is the destination operand. The condition *COND* determining whether the copy operation takes place is based on some combination of condition code values, similar to the test and conditional jump instructions. As an example, the `cmovll` instruction performs a copy when the condition codes indicate a value less than zero. Note that the first '`l`' of this instruction indicates "less," while the second is the GAS suffix for long word.

The following assembly code shows how to implement absolute value with conditional move.

```
1   movl 8(%ebp),%eax              Get val as result
2   movl %eax,%edx                 Copy to %edx
3   negl %edx                      Negate %edx
4   testl %eax,%eax                Test val
    Conditionally move %edx to %eax
5   cmovll %edx,%eax               If < 0, copy %edx to result
```

As this code shows, the strategy is to set `val` as a return value, compute `-val`, and conditionally move it to register `%eax` to change the return value when `val` is negative. Our measurements of this code shows that it runs for 13.7 cycles regardless of the data patterns. This clearly yields better overall performance than a procedure that requires between 13 and 27 cycles.

Practice Problem 5.5:

A friend of yours has written an optimizing compiler that makes use of conditional move instructions. You try compiling the following C code:

```
1 /* Dereference pointer or return 0 if null */
2 int deref(int *xp)
3 {
4     return xp ? *xp : 0;
5 }
```

The compiler generates the following code for the body of the procedure.

```
1   movl 8(%ebp),%edx          Get xp
2   movl (%edx),%eax           Get *xp as result
3   testl %edx,%edx            Test xp
4   cmovll %edx,%eax           If 0, copy 0 to result
```

Explain why this code does not provide a valid implementation of `deref`

The current version of GCC does not generate any code using conditional moves. Due to a desire to remain compatible with earlier 486 and Pentium processors, the compiler does not take advantage of these new features. In our experiments, we used the handwritten assembly code shown above. A version using GCC's facility to embed assembly code within a C program (Section 3.15) required 17.1 cycles due to poorer quality code generation.

Unfortunately, there is not much a C programmer can do to improve the branch performance of a program, except to recognize that data-dependent branches incur a high cost in terms of performance. Beyond this, the programmer has little control over the detailed branch structure generated by the compiler, and it is hard to make branches more predictable. Ultimately, we must rely on a combination of good code generation by the compiler to minimize the use of conditional branches, and effective branch prediction by the processor to reduce the number of branch mispredictions.

5.13 Understanding Memory Performance

All of the code we have written, and all the tests we have run, require relatively small amounts of memory. For example, the combining routines were measured over vectors of length 1024, requiring no more than 8,096 bytes of data. All modern processors contain one or more *cache* memories to provide fast access to such small amounts of memory. All of the timings in Figure 5.12 assume that the data being read or written

code/opt/list.c

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

code/opt/list.c

Figure 5.30: **Linked List Functions.** These illustrate the latency of the load operation.

is contained in cache. In Chapter 6, we go into much more detail about how caches work and how to write code that makes best use of the cache.

In this section, we will further investigate the performance of load and store operations while maintaining the assumption that the data being read or written are held in cache. As Figure 5.12 shows, both of these units have a latency of 3, and an issue time of 1. All of our programs so far have used only load operations, and they have had the property that the address of one load depended on incrementing some register, rather than as the result of another load. Thus, as shown in Figures 5.15 to 5.18, 5.21 and 5.26, the load operations could take advantage of pipelining to initiate new load operations on every cycle. The relatively long latency of the load operation has not had any adverse affect on program performance.

5.13.1 Load Latency

As an example of code whose performance is constrained by the latency of the load operation, consider the function `list_len`, shown in Figure 5.30. This function 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 a CPE of 3.00, which we claim is a direct reflection of the latency of the load operation. To see this, consider the assembly code for the loop, and the translation of its first iteration into operations:

Assembly Instructions	Execution Unit Operations
`.L27:`	
`incl %eax`	`incl %eax.0` → `%eax.1`
`movl (%edx),%edx`	`load (%edx.0)` → `%edx.1`
`testl %edx,%edx`	`testl %edx.1,%edx.1` → `cc.1`
`jne .L27`	`jne-taken cc.1`

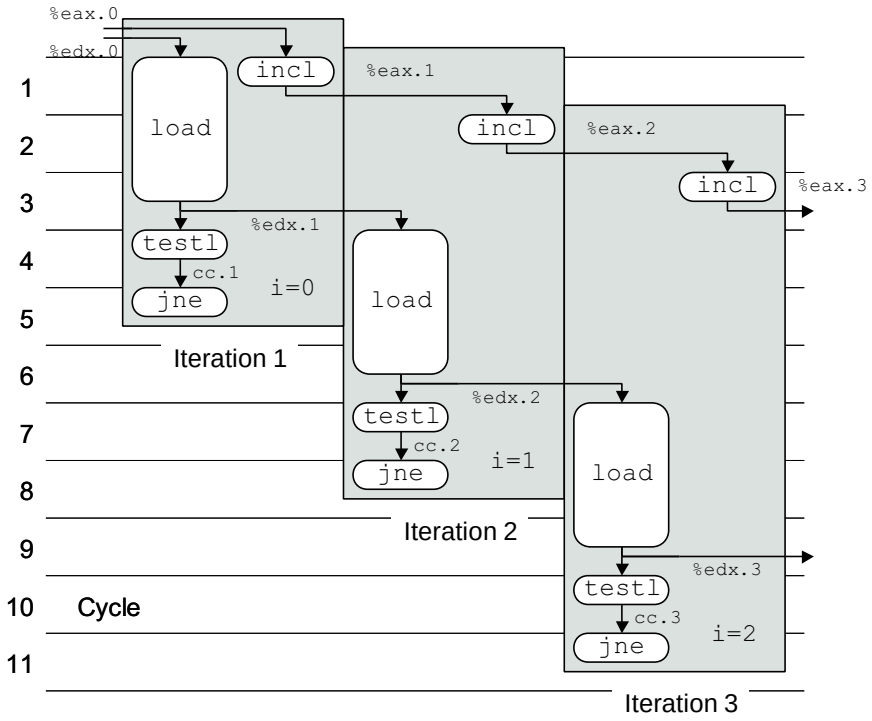


Figure 5.31: **Scheduling of Operations for List Length Function.** The latency of the load operation limits the CPE to a minimum of 3.0.

code/opt/copy.c

```
1 /* Set element of array to 0 */
2 static void array_clear(int *src, int *dest, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         dest[i] = 0;
8 }
9
10 /* Set elements of array to 0, unrolling by 8 */
11 static void array_clear_8(int *src, int *dest, int n)
12 {
13     int i;
14     int len = n - 7;
15
16     for (i = 0; i < len; i+=8) {
17         dest[i] = 0;
18         dest[i+1] = 0;
19         dest[i+2] = 0;
20         dest[i+3] = 0;
21         dest[i+4] = 0;
22         dest[i+5] = 0;
23         dest[i+6] = 0;
24         dest[i+7] = 0;
25     }
26     for (; i < n; i++)
27         dest[i] = 0;
28 }
```

code/opt/copy.c

Figure 5.32: **Functions to Clear Array.** These illustrate the pipelining of the store operation.

Each successive value of register `%edx` depends on the result of a load operation having `%edx` as an operand. Figure 5.31 shows the scheduling of operations for the first three iterations of this function. As can be seen, the latency of the load operation limits the CPE to 3.0.

5.13.2 Store Latency

In all of our examples so far, we have interacted with the memory only by using the load operation to read from a memory location into a register. Its counterpart, the *store* operation, writes a register value to memory. As Figure 5.12 indicates, this operation also has a nominal latency of three cycles, and an issue time of one cycle. However, its behavior, and its interactions with load operations, involve several subtle issues.

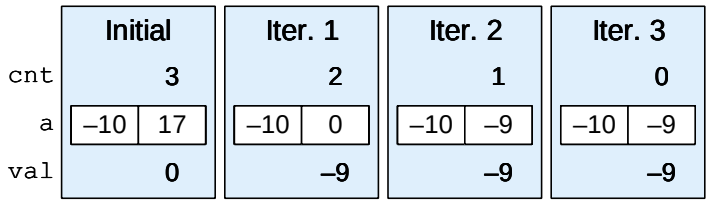
As with the load operation, in most cases the store operation can operate in a fully pipelined mode, beginning

code/opt/copy.c

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

code/opt/copy.c

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



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

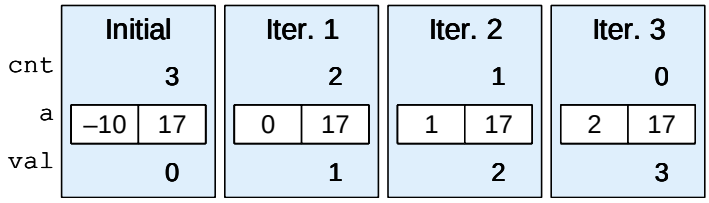


Figure 5.33: **Code to Write and Read Memory Locations, Along with Illustrative Executions.** This function highlights the interactions between stores and loads when arguments `src` and `dest` are equal.

a new store on every cycle. For example, consider the functions shown in Figure 5.32 that set the elements of an array `dest` of length `n` to zero. Our measurements for the first version show a CPE of 2.00. Since each iteration requires a store operation, it is clear that the processor can begin a new store operation at least once every two cycles. To probe further, we try unrolling the loop eight times, as shown in the code for `array_clear_8`. For this one we measure a CPE of 1.25. That is, each iteration requires around ten cycles and issues eight store operations. Thus, we have nearly achieved the optimum limit of one new store operation per cycle.

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 must be independent from each other. In fact, only a load operation is affected by the result of a store operation, since only a load can read back the memory location that has been written by the store. The function `write_read` shown in Figure 5.33 illustrates the potential interactions between loads and stores. This figure also shows two example executions of this

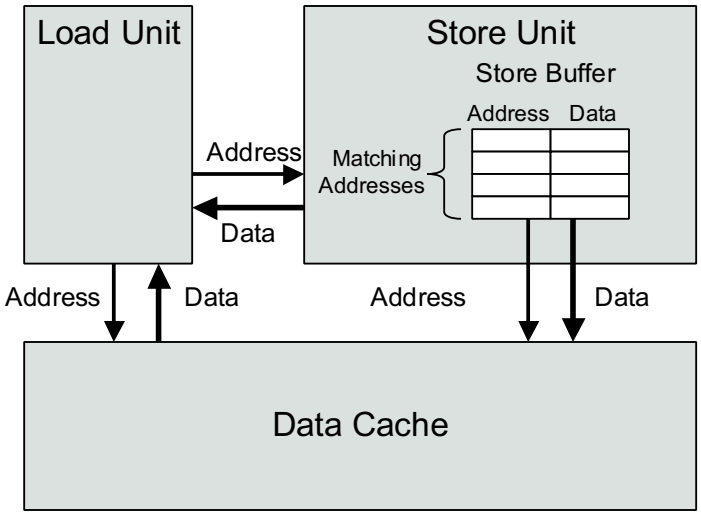


Figure 5.34: **Detail of Load and Store Units.** The store unit maintains a buffer of pending writes. The load unit must check its address with those in the store unit to detect a write/read dependency.

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, but over a larger number of iterations, gives a CPE of 2.00.

In example B of Figure 5.33(a), 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`. 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 *write/read dependency*—the outcome of a memory read depends on a very recent memory write. Our performance measurements show that example B has a CPE of 6.00. The write/read dependency causes a slowdown in the processing.

To see how the processor can distinguish between these two cases and why one runs slower than another, we must take a more detailed look at the load and store execution units, as shown in Figure 5.34. The store unit contains 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, it retrieves the corresponding data entry as the result of the load operation.

The assembly code for the inner loop, and its translation into operations during the first iteration, is as follows:

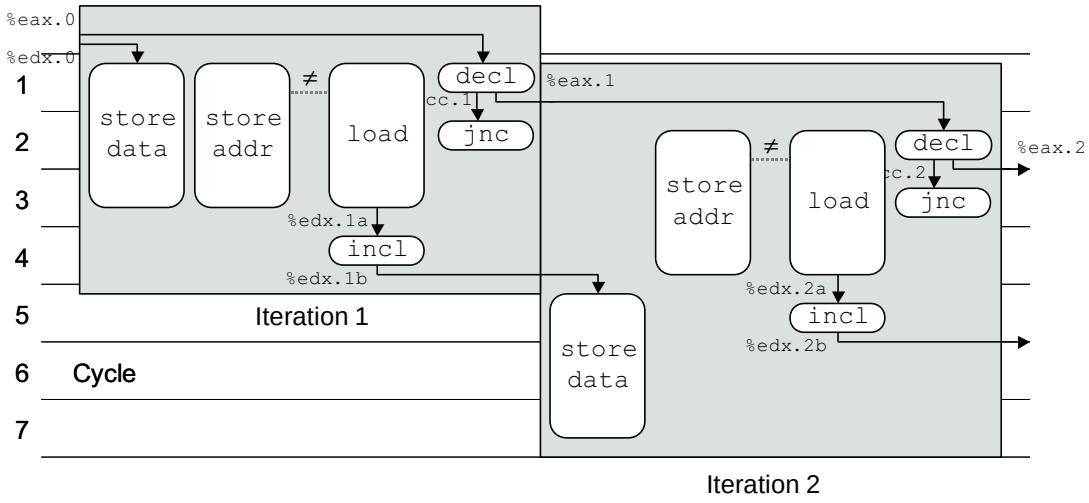


Figure 5.35: **Timing of `write_read` for Example A.** The store and load operations have different addresses, and so the load can proceed without waiting for the store.

Assembly Instructions	Execution Unit Operations
`.L32:`	
`movl %edx,(%ecx)`	`storeaddr (%ecx)`
	`storedata %edx.0`
`movl (%ebx),%edx`	`load (%ebx)` → `%edx.1a`
`incl %edx`	`incl %edx.1a` → `%edx.1b`
`decl %eax`	`decl $eax.0` → `%eax.1`
`jnc .L32`	`jnc-taken cc.1`

Observe that the instruction `movl %edx,(%ecx)` is translated into two operations: the `storeaddr` instruction computes the address for the store operation, creates an entry in the store buffer, and sets the address field for that entry. The `storedata` instruction sets the data field for the entry. Since there is only one store unit, and store operations are processed in program order, there is no ambiguity about how the two operations match up. As we will see, the fact that these two computations are performed independently can be important to program performance.

Figure 5.35 shows the timing of the operations for the first two iterations of `write_read` for the case of example A. As indicated by the dotted line between the `storeaddr` and `load` operations, the `store-addr` operation creates an entry in the store buffer, which is then checked by the `load`. Since these are unequal, the load proceeds to read the data from the cache. Even though the store operation has not been completed, the processor can detect that it will affect a different memory location than the load is trying to read. This process is repeated on the second iteration as well. Here we can see that the `storedata` operation must wait until the result from the previous iteration has been loaded and incremented. Long before this, the `storeaddr` operation and the `load` operations can match up their adddresses, determine they are different, and allow the load to proceed. In our computation graph, we show the load for the second iteration beginning just one cycle after the load from the first. If continued for more iterations, we would find the

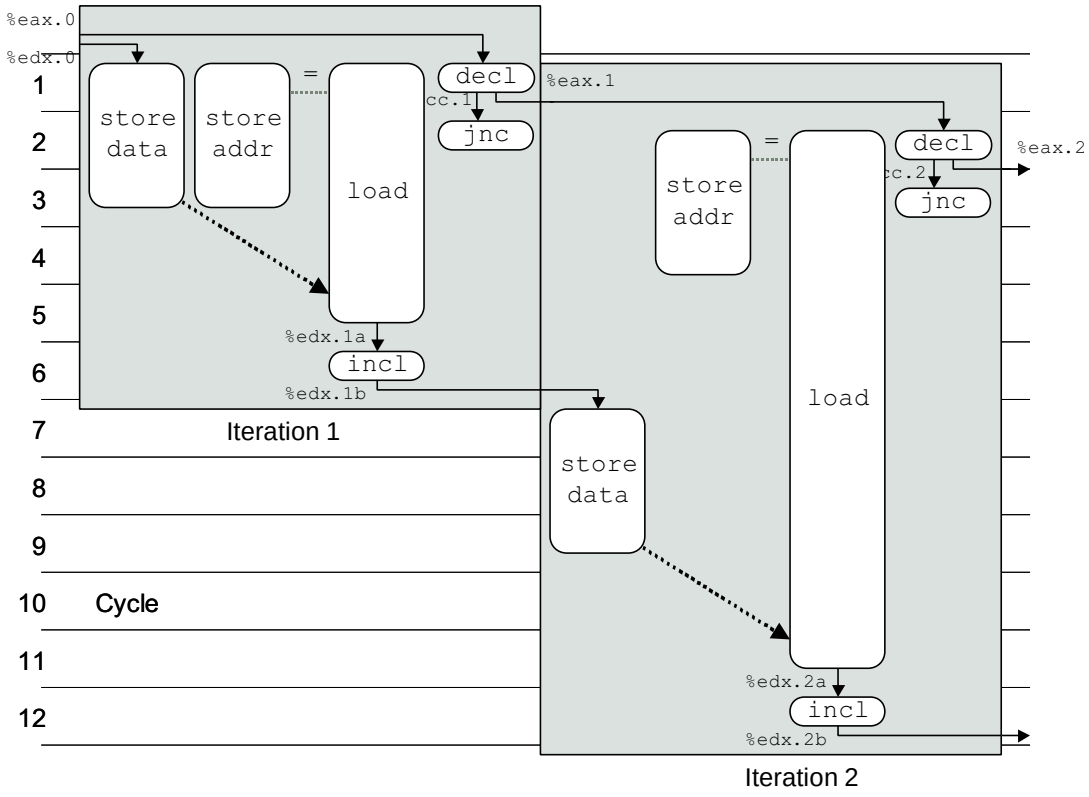


Figure 5.36: **Timing of `write_read` for Example B.** The store and load operations have the same address, and hence the load must wait until it can get the result from the store.

graph indicates a CPE of 1.0. Evidentally, some other resource constraint limits the actual performance to a CPE of 2.0.

Figure 5.36 shows the timing of the operations for the first two iterations of `write_read` for the case of example B. Again, the dotted line between the `storeaddr` and `load` operations indicates that the the `storeaddr` operation creates an entry in the store buffer which is then checked by the `load`. Since these are equal, the load must wait until the `storedata` operation has completed, and then it gets the data from the store buffer. This waiting is indicated in the graph by a much more elongated box for the `load` operation. In addition, we show a dashed arrow from the `storedata` to the `load` operations to indicate that the result of the `storedata` is passed to the `load` as its result. Our timings of these operations are drawn to reflect the measured CPE of 6.0. Exactly how this timing arises is not totally clear, however, and so these figures are intended to be more illustrative than factual. In general, the processor/memory interface is one of the most complex portions of a processor design. Without access to detailed documentation and machine analysis tools, we can only give a hypothetical description of the actual behavior.

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. Since memory

operations make up a significant fraction of the program, the memory subsystem is optimized to run with greater parallelism for independent memory operations.

Practice Problem 5.6:

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

```
1 static void copy_array(int *src, int *dest, int n)
2 {
3     int i;
4
5     for (i = 0; i < n; i++)
6         dest[i] = src[i];
7 }
```

Suppose `a` is an array of length 1000 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 3.00, while the call of part B has a CPE of 5.00. To what factor do you attribute this performance difference?

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

5.14 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:

1. 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.

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

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

   (b) 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.

3. Low-level optimizations.

   (a) Try various forms of pointer versus array code.

   (b) Reduce loop overhead by unrolling loops.

   (c) Find ways to make use of the pipelined functional units by techniques such as iteration splitting.

A final word of advice to the reader is to be careful to avoid expending effort on misleading results. One useful technique is to use checking code to test each version of the code as it is being optimized to make sure no bugs are introduced during this process. Checking code applies a series of tests to the program and makes sure it obtains the desired results. It is very easy to make mistakes when one is introducing new variables, changing loop bounds, and making the code more complex overall. In addition, it is important to notice any unusual or unexpected changes in performance. As we have shown, the selection of the benchmark data can make a big difference in performance comparisons due to performance anomalies, and because we are only executing short instruction sequences.

5.15 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 requires optimization. When working with large programs, even knowing where to focus our optimizations 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 present a general principle of system optimization known as *Amdahl's Law*.

5.15.1 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 on which we should focus 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. We show this for a C program `prog.c`, to be running 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:

   ```
   unix> gcc -O2 -pg prog.c -o prog
   ```

2. The program is then executed as usual:

   ```
   unix> ./prog file.txt
   ```

   It runs slightly (up to a factor of two) 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`.

```
unix> 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 shows this part of the report for the first three functions in a program:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 85.62      7.80     7.80        1  7800.00  7800.00  sort_words
  6.59      8.40     0.60   946596     0.00     0.00  find_ele_rec
  4.50      8.81     0.41   946596     0.00     0.00  lower1
```

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 7.80 seconds, while the function `lower1` was called 946,596 times, requiring a total of 0.41 seconds.

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

```
                                4872758             find_ele_rec [5]
                0.60    0.01  946596/946596         insert_string [4]
[5]      6.7    0.60    0.01  946596+4872758 find_ele_rec [5]
                0.00    0.01   26946/26946          save_string [9]
                0.00    0.00   26946/26946          new_ele [11]
                                4872758             find_ele_rec [5]
```

This history shows both the functions that called `find_ele_rec`, as well as the functions that it called. In the upper part, we find that the function was actually called 5,819,354 times (shown as "`946596+4872758`")—4,872,758 times by itself, and 946,596 times by function `insert_string` (which itself was called 946,596 times). Function `find_ele_rec` in turn called two other functions: `save_string` and `new_ele`, each a total of 26,946 times.

From this calling information, we can often infer useful information about the program behavior. For example, the function `find_ele_rec` is a recursive procedure that scans a linked list looking for a particular string. Given that the ratio of recursive to top-level calls was 5.15, we can infer that it required scanning an average of around 6 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, as will be discussed in Chapter 9. In brief, 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 δ. Typical values of δ range between 1.0 and 10.0 milliseconds. It then determines what function the program was executing when the interrupt occurs and increments the

counter for that function by δ. 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 one second, however, the numbers should be viewed as only rough estimates.

- The calling information is quite reliable. 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.

5.15.2 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 reads a text file, creates a table of unique words and how many times each word occurs, and then sorts the words in descending order of occurrence. As a benchmark, we ran it on a file consisting of the complete works of William Shakespeare. From this, we determined that Shakespeare wrote a total of 946,596 words, of which 26,946 are unique. The most common word was "the," occurring 29,801 times. The word "love" occurs 2249 times, while "death" occurs 933.

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

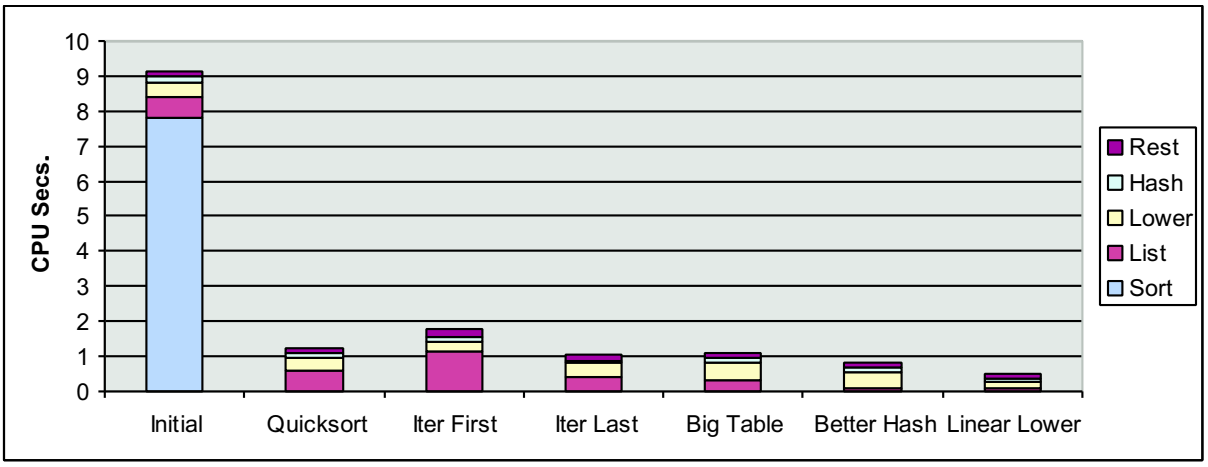
1. Each word is read from the file and converted to lower case. Our initial version used the function `lower1` (Figure 5.7), which we know to have quadratic complexity.
2. A hash function is applied to the string to create a number between 0 and $s - 1$, for a hash table with s buckets. Our initial function 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 word 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 5.37 shows the profile results for different versions of our word-frequency analysis program. For each version, we divide the time into five categories:

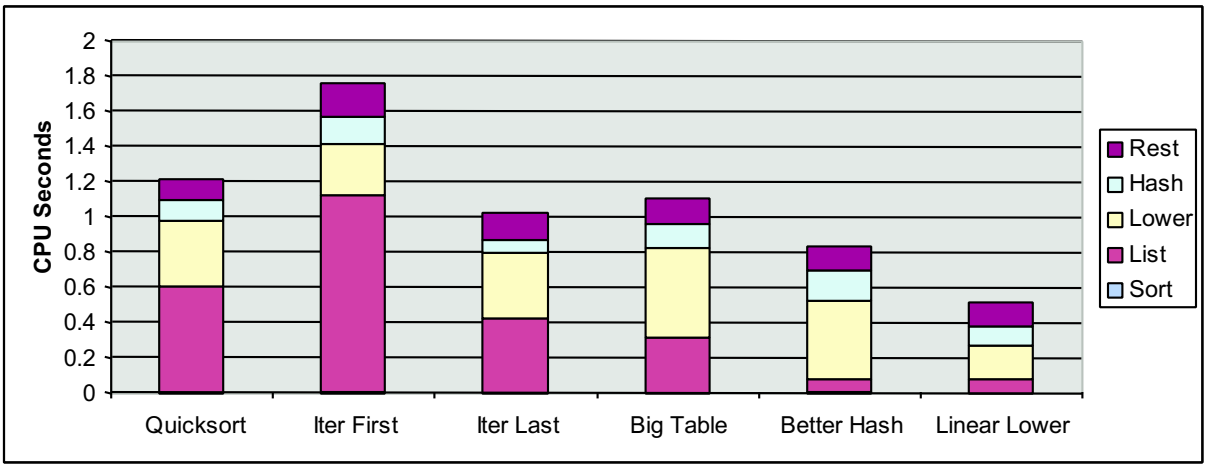
Sort Sorting the words by frequency.

List Scanning the linked list for a matching word, inserting a new element if necessary.

Lower Converting the string to lower case.



(a) All versions.



(b) All but the slowest version.

Figure 5.37: **Profile Results for Different Version of Word 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 requires over 9 seconds, with most of the time spent sorting. This is not surprising, since insertion sort has quadratic complexity, and the program sorted nearly 27,000 values.

In our next version, we performed sorting using the library function `qsort`, which is based on the quicksort algorithm. 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 1.2 seconds. Part (b) of the figure shows the times for the remaining version on a scale where we can see them better.

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 1.8 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 words to occur near the beginnings of the lists. That way, the function will quickly locate the common cases. Assuming words are spread uniformly throughout the document, we would expect the first occurrence of a frequent one come before that of a less frequent one. By inserting new words at the end, the first function tended to order words 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 1.0 seconds, just slightly better than with the recursive version.

Next, we consider the hash table structure. The initial version had only 1021 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 26,946 entries, this would imply an average *load* of $26946/1007 = 26.4$. That explains why so much of the time is spent performing list operations—the searches involve testing a significant number of candidate words. It also explains why the performance is so sensitive to the list ordering. We then increased the number of buckets to 10,007, reducing the average load to 2.70. Oddly enough, however, our overall run time increased to 1.11 seconds. The profile results indicate that this additional time was mostly spent with the lower-case conversion routine, although this is highly unlikely. Our run times are sufficiently short that we cannot expect very high accuracy with these timings.

We hypothesized that the poor performance with a larger table was due to a poor choice of hash function. Simply summing the character codes does not produce a very wide range of values and does not differentiate according to the ordering of the characters. For example, the words "god" and "dog" would hash to location $147 + 157 + 144 = 448$, since they contain the same characters. The word "foe" would also hash to this location, since $146 + 157 + 145 = 448$. We switched to a hash function that uses shift and EXCLUSIVE-OR operations. With this version, shown as "Better Hash," the time drops to 0.84 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 one half of the time is spent performing lower-case conversion. We have already seen that function `lower1` has very poor performance, especially for long strings. The words in this document are short enough to avoid the disasterous consequences of quadratic pe-

formance; the longest word ("honorificabilitudinitatibus") is 27 characters long. Still, switching to `lower2`, shown as "Linear Lower" yields a significant performance, with the overall time dropping to 0.52 seconds.

With this exercise, we have shown that code profiling can help drop the time required for a simple application from 9.11 seconds down to 0.52—a factor of 17.5 improvement. The profiler helps us focus our attentionon the most time-consuming parts of the program and also provides useful information about the procedure call structure.

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 (under one second) run times. 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 lower-case conversion routine was the major performance bottleneck. Even worse, if only profiled documents with short words, we might never never detect hidden performance killers 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 is mainly involves avoiding algorithms (such as insertion sort) and bad programming practices (such as `lower1`) that yield poor asymptotic performance.

5.15.3 Amdahl's Law

Gene Amdahl, one of the early pioneers in computing, made a simple, but insightful observation about the effectiveness of improving the performance of one part of a system. This observation is therefore called *Amdahl's Law*. The main idea is that when we speed up one part of a system, the effect on the overall system performance depends on both how significant this part was and how much it sped up. Consider a system where executing some application requires time T_{old}. Suppose, some part of the system requires a fraction α of this time, and that we improve its performance by a factor of k. That is, the component originally required time αT_{old}, and it now requires time $(\alpha T_{old})/k$. The overall execution time will be:

$$\begin{aligned} T_{new} &= (1-\alpha)T_{old} + (\alpha T_{old})/k \\ &= T_{old}[(1-\alpha) + \alpha/k]. \end{aligned}$$

From this, we can compute the speedup $S = T_{old}/T_{new}$ as:

$$S = \frac{1}{(1-\alpha) + \alpha/k} \tag{5.1}$$

As an example, consider the case where a part of the system that initially consumed 60% of the time ($\alpha = 0.6$) is sped up by a factor of 3 ($k = 10$). Then we get a speedup of $1/[0.4 + 0.6/3] = 1.67$. Thus, even though we made a substantial improvement to a major part of the system, our net speedup was significantly less. This is the major insight of Amdahl's Law—to significantly speed up the entire system, we must improve the speed of a very large fraction of the overall system.

Practice Problem 5.7:

The marketing department at your company has promised your customers that the next software release will show a 2X performance improvement. You have been assigned the task of delivering on that

promise. You have determined that only 80% of the system can be improved. How much (i.e., what value of k) would you need to improve this part to meet the overall performance target?

One interesting special case of Amdahl's Law is to consider the case where $k = \infty$. That is, we are able to take some part of the system and speed it up to the point where it takes a negligible amount of time. We then get

$$S_\infty = \frac{1}{(1-\alpha)} \tag{5.2}$$

So, for example, if we can speed up 60% of the system to the point where it requires close to no time, our net speedup will still only be $1/0.4 = 2.5$. We saw this performance with our dictionary program as we replaced insertion sort by quicksort. The initial version spent 7.8 of its 9.1 seconds performing insertion sort, giving $\alpha = .86$. With quicksort, the time spent sorting becomes negligible, giving a predicted speedup of 7.1. In fact the actual speedup was higher: $9.11/1.22 = 7.5$, due to inaccuracies in the profiling measurements for the initial version. We were able to gain a large speedup because sorting constituted a very large fraction of the overall execution time.

Amdahl's Law describes a general principle for improving any process. In addition to applying to speeding up computer systems, it can guide company trying to reduce the cost of manufacturing razor blades, or to a student trying to improve his or her gradepoint average. Perhaps it is most meaningful in the world of computers, where we routinely improve performance by factors of two or more. Such high factors can only be obtained by optimizing a large part of the system.

5.16 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 have also see 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 in eliminating these.

Beyond this, we have studied a series of techniques, including loop unrolling, iteration splitting, and pointer arithmetic. 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. For execution on a modern, out-of-order processor, much can be gained by analyzing how the program would execute on a machine with unlimited processing resources, but where the latencies and the issue times of the functional units match those of the target processor. To refine this analysis, we should also consider such resource constraints as the number and types of functional units.

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 loops more predictable and to try to reduce interactions between store and load operations.

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. These tools can break down the execution time below the procedure level, to measure performance of each *basic block* of the program. A basic block is a sequence of instructions with no conditional operations.

Amdahl's Law provides a simple, but powerful insight into the performance gains obtained by improving just one part of the system. The gain depends both on how much we improve this part and how large a fraction of the overall time this part originally required.

Bibliographic Notes

Many books have been written about compiler optimization techniques. Muchnick's book is considered the most comprehensive [52]. Wadleigh and Crawford's book on software optimization [81] covers some of the material we have, but also describes the process of getting high performance on parallel machines.

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 [31, Ch. 4]. Shriver and Smith give a detailed presentation of an AMD processor [65] that bears many similarities to the one we have described.

Amdahl's Law is presented in most books on computer architecture. With its major focus on quantitative system evaluation, Hennessy and Patterson's book [31] provides a particularly good treatment.

Homework Problems

Homework Problem 5.8 [Category 2]:

Suppose that we wish to write a procedure that computes the inner product of two vectors. An abstract version of the function has a CPE of 54 for both 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  /* Accumulate in temporary */
2  void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3  {
4      int i;
5      int 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 requires 3.11 cycles per iteration for integer data. The assembly code for the inner loop is:

```
     udata in %esi, vdata in %ebx, i in %edx, sum in %ecx, length in %edi
1 .L24:                                  loop:
2   movl (%esi,%edx,4),%eax                Get udata[i]
3   imull (%ebx,%edx,4),%eax               Multiply by vdata[i]
4   addl %eax,%ecx                         Add to sum
5   incl %edx                              i++
6   cmpl %edi,%edx                         Compare i:length
7   jl .L24                                If <, goto loop
```

Assume that integer multiplication is performed by the general integer functional unit and that this unit is pipelined. This means that one cycle after a multiplication has started, a new integer operation (multiplication or otherwise) can begin. Assume also that the Integer/Branch function unit can perform simple integer operations.

A. Show a translation of these lines of assembly code into a sequence of operations. The `movl` instruction translates into a single `load` operation. Register `%eax` gets updated twice in the loop. Label the different versions `%eax.1a` and `%eax.1b`.

B. Explain how the function can go faster than the number of cycles required for integer multiplication.

C. Explain what factor limits the performance of this code to at best a CPE of 2.5.

D. For floating-point data, we get a CPE of 3.5. Without needing to examine the assembly code, describe a factor that will limit the performance to at best 3 cycles per iteration.

Homework Problem 5.9 [Category 1]:

Write a version of the inner product procedure described in Problem 5.8 that uses four-way loop unrolling.

Our measurements for this procedure give a CPE of 2.20 for integer data and 3.50 for floating point.

A. Explain why any version of any inner product procedure cannot achieve a CPE better than 2.

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

Homework Problem 5.10 [Category 1]:

Write a version of the inner product procedure described in Problem 5.8 that uses four-way loop unrolling and two-way parallelism.

Our measurements for this procedure give a CPE of 2.25 for floating-point data. Describe two factors that limit the performance to a CPE of at best 2.0.

Homework Problem 5.11 [Category 2]:

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iteration:

```
1 int fact(int n)
2 {
3     int i;
4     int result = 1;
5
6     for (i = n; i > 0; i--)
7         result = result * i;
8     return result;
9 }
```

By doing so, they have reduced the number of CPE for the function from 63 to 4, measured on an Intel Pentium III (really!). Still, they would like to do better.

One of the programmers heard about loop unrolling. She generated the following code:

```
1 int fact_u2(int n)
2 {
3     int i;
4     int result = 1;
5     for (i = n; i > 0; i-=2) {
6         result = (result * i) * (i-1);
7     }
8     return result;
9 }
```

Unfortunately, the team discovered that this code returns 0 for some values of argument `n`.

A. For what values of `n` will `fact_u2` and `fact` return different values?

B. Show how to fix `fact_u2`. Note that there is a special trick for this procedure that involves just changing a loop bound.

C. Benchmarking `fact_u2` shows no improvement in performance. How would you explain that?

D. You modify the line inside the loop to read:

```
7         result = result * (i * (i - 1));
```

To everyone's astonishment, the measured performance now has a CPE of 2.5. How do you explain this performance improvement?

Homework Problem 5.12 [Category 1]:

Using the conditional move instruction, write assembly code for the body of the following function:

```
1 /* Return maximum of x and y */
2 int max(int x, int y)
3 {
4     return (x < y) ? y : x;
5 }
```

Homework Problem 5.13 [Category 2]:

Using conditional moves, the general technique for translating a statement of the form:

`val =` *cond-expr* `?` *then-expr* `:` *else-expr*`;`

is to generate code of the form:

```
val = then-expr;
temp = else-expr;
test = cond-expr;
if (test) val = temp;
```

where the last line is implemented with a conditional move instruction. Using the example of Practice Problem 5.5 as a guide, state the general requirements for this translation to be valid.

Homework Problem 5.14 [Category 2]:

The following function computes the sum of the elements in a linked list:

```
1 static int list_sum(list_ptr ls)
2 {
3     int sum = 0;
4
5     for (; ls; ls = ls->next)
6         sum += ls->data;
7     return sum;
8 }
```

The assembly code for the loop, and its translation of the first iteration into operations yields the following:

Assembly Instructions	Execution Unit Operations
`.L43:`	
`addl 4(%edx),%eax`	`movl 4(%edx.0)` → `t.1`
	`addl t.1,%eax.0` → `%eax.1`
`movl (%edx),%edx`	`load (%edx.0)` → `%edx.1`
`testl %edx,%edx`	`testl %edx.1,%edx.1` → `cc.1`
`jne .L43`	`jne-taken cc.1`

A. Draw a graph showing the scheduling of operations for the first three iterations of the loop, in the style of Figure 5.31. Recall that there is just one load unit.

B. Our measurements for this function give a CPE of 4.00. Is this consistent with the graph you drew in part A?

Homework Problem 5.15 [Category 2]:

The following function is a variant on the list sum function shown in Problem 5.14:

```
 1 static int list_sum2(list_ptr ls)
 2 {
 3     int sum = 0;
 4     list_ptr old;
 5
 6     while (ls) {
 7         old = ls;
 8         ls = ls->next;
 9         sum += old->data;
10     }
11     return sum;
12 }
```

This code is written in such a way that the memory access to fetch the next list element comes before the one to retrieve the data field from the current element.

The assembly code for the loop, and its translation of the first iteration into operations yields the following:

Assembly Instructions	Execution Unit Operations
`.L48:`	
`movl %edx,%ecx`	
`movl (%edx),%edx`	`load (%edx.0)` → `%edx.1`
`addl 4(%ecx),%eax`	`movl 4(%edx.0)` → `t.1`
	`addl t.1,%eax.0` → `%eax.1`
`testl %edx,%edx`	`testl %edx.1,%edx.1` → `cc.1`
`jne .L48`	`jne-taken cc.1`

Note that the register move operation `movl %edx,%ecx` does not require any operations to implement. It is handled by simply associating the tag `edx.0` with register `%ecx`, so that the later instruction `addl 4(%ecx),%eax` is translated to use `edx.0` as its source operand.

A. Draw a graph showing the scheduling of operations for the first three iterations of the loop, in the style of Figure 5.31. Recall that there is just one load unit.

B. Our measurements for this function give a CPE of 3.00. Is this consistent with the graph you drew in part A?

C. How does this function make better use of the load unit than did the function of Problem 5.14?