CHAPTER 5

Optimizing Program Performance

5.1 Capabilities and Limitations of Optimizing Compilers 476
5.2 Expressing Program Performance 480
5.3 Program Example 482
5.4 Eliminating Loop Inefficiencies 486
5.5 Reducing Procedure Calls 490
5.6 Eliminating Unneeded Memory References 491
5.7 Understanding Modern Processors 496
5.8 Loop Unrolling 509
5.9 Enhancing Parallelism 513
5.10 Summary of Results for Optimizing Combining Code 524
5.11 Some Limiting Factors 525
5.12 Understanding Memory Performance 531
5.13 Life in the Real World: Performance Improvement Techniques 539
5.14 Identifying and Eliminating Performance Bottlenecks 540
5.15 Summary 547

Bibliographic Notes 548
Homework Problems 549
Solutions to Practice Problems 552
The biggest speedup you’ll ever get with a program will be when you first get it working.

—John K. Ousterhout

The primary objective in writing a program must be to make it work correctly under all possible conditions. A program that runs fast but gives incorrect results serves no useful purpose. Programmers must write clear and concise code, not only so that they can make sense of it, but also so that others can read and understand the code during code reviews and when modifications are required later.

On the other hand, there are many occasions when making a program run fast is also an important consideration. If a program must process video frames or network packets in real time, then a slow-running program will not provide the needed functionality. When a computation task is so demanding that it requires days or weeks to execute, then making it run just 20% faster can have significant impact. In this chapter, we will explore how to make programs run faster via several different types of program optimization.

Writing an efficient program requires several types of activities. First, we must select an appropriate set of algorithms and data structures. Second, we must write source code that the compiler can effectively optimize to turn into efficient executable code. For this second part, it is important to understand the capabilities and limitations of optimizing compilers. Seemingly minor changes in how a program is written can make large differences in how well a compiler can optimize it. Some programming languages are more easily optimized than others. Some features of C, such as the ability to perform pointer arithmetic and casting, make it challenging for a compiler to optimize. Programmers can often write their programs in ways that make it easier for compilers to generate efficient code. A third technique for dealing with especially demanding computations is to divide a task into portions that can be computed in parallel, on some combination of multiple cores and multiple processors. We will defer this aspect of performance enhancement to Chapter 12. Even when exploiting parallelism, it is important that each parallel thread execute with maximum performance, and so the material of this chapter remains relevant in any case.

In approaching program development and optimization, we must consider how the code will be used and what critical factors affect it. In general, programmers must make a trade-off between how easy a program is to implement and maintain, and how fast it runs. At an algorithmic level, a simple insertion sort can be programmed in a matter of minutes, whereas a highly efficient sort routine may take a day or more to implement and optimize. At the coding level, many low-level optimizations tend to reduce code readability and modularity, making the programs more susceptible to bugs and more difficult to modify or extend. For code that will be executed repeatedly in a performance-critical environment, extensive optimization may be appropriate. One challenge is to maintain some degree of elegance and readability in the code despite extensive transformations.

We describe a number of techniques for improving code performance. Ideally, a compiler would be able to take whatever code we write and generate the most
efficient possible machine-level program having the specified behavior. Modern compilers employ sophisticated forms of analysis and optimization, and they keep getting better. Even the best compilers, however, can be thwarted by optimization blockers—aspects of the program’s behavior that depend strongly on the execution environment. Programmers must assist the compiler by writing code that can be optimized readily.

The first step in optimizing a program is to eliminate unnecessary work, making the code perform its intended task as efficiently as possible. This includes eliminating unnecessary function calls, conditional tests, and memory references. These optimizations do not depend on any specific properties of the target machine.

To maximize the performance of a program, both the programmer and the compiler require a model of the target machine, specifying how instructions are processed and the timing characteristics of the different operations. For example, the compiler must know timing information to be able to decide whether it should use a multiply instruction or some combination of shifts and adds. Modern computers use sophisticated techniques to process a machine-level program, executing many instructions in parallel and possibly in a different order than they appear in the program. Programmers must understand how these processors work to be able to tune their programs for maximum speed. We present a high-level model of such a machine based on recent designs of Intel and AMD processors. We also devise a graphical data-flow notation to visualize the execution of instructions by the processor, with which we can predict program performance.

With this understanding of processor operation, we can take a second step in program optimization, exploiting the capability of processors to provide instruction-level parallelism, executing multiple instructions simultaneously. We cover several program transformations that reduce the data dependencies between different parts of a computation, increasing the degree of parallelism with which they can be executed.

We conclude the chapter by discussing issues related to optimizing large programs. We describe the use of code profilers—tools that measure the performance of different parts of a program. This analysis can help find inefficiencies in the code and identify the parts of the program on which we should focus our optimization efforts. Finally, we present an important observation, known as Amdahl’s law, which quantifies the overall effect of optimizing some portion of a system.

In this presentation, we make code optimization look like a simple linear process of applying a series of transformations to the code in a particular order. In fact, the task is not nearly so straightforward. A fair amount of trial-and-error experimentation is required. This is especially true as we approach the later optimization stages, where seemingly small changes can cause major changes in performance, while some very promising techniques prove ineffective. As we will see in the examples that follow, it can be difficult to explain exactly why a particular code sequence has a particular execution time. Performance can depend on many detailed features of the processor design for which we have relatively little documentation or understanding. This is another reason to try a number of different variations and combinations of techniques.
Studying the assembly-code representation of a program is one of the most effective means for gaining an understanding of the compiler and how the generated code will run. A good strategy is to start by looking carefully at the code for the inner loops, identifying performance-reducing attributes such as excessive memory references and poor use of registers. Starting with the assembly code, we can also predict what operations will be performed in parallel and how well they will use the processor resources. As we will see, we can often determine the time (or at least a lower bound on the time) required to execute a loop by identifying critical paths, chains of data dependencies that form during repeated executions of a loop. We can then go back and modify the source code to try to steer the compiler toward more efficient implementations.

Most major compilers, including gcc, are continually being updated and improved, especially in terms of their optimization abilities. One useful strategy is to do only as much rewriting of a program as is required to get it to the point where the compiler can then generate efficient code. By this means, we avoid compromising the readability, modularity, and portability of the code as much as if we had to work with a compiler of only minimal capabilities. Again, it helps to iteratively modify the code and analyze its performance both through measurements and by examining the generated assembly code.

To novice programmers, it might seem strange to keep modifying the source code in an attempt to coax the compiler into generating efficient code, but this is indeed how many high-performance programs are written. Compared to the alternative of writing code in assembly language, this indirect approach has the advantage that the resulting code will still run on other machines, although perhaps not with peak performance.

5.1 Capabilities and Limitations of Optimizing Compilers

Modern compilers employ sophisticated algorithms to determine what values are computed in a program and how they are used. They can then exploit opportunities to simplify expressions, to use a single computation in several different places, and to reduce the number of times a given computation must be performed. Most compilers, including gcc, provide users with some control over which optimizations they apply. As discussed in Chapter 3, the simplest control is to specify the optimization level. For example, invoking gcc with the command-line flag ‘-O1’ will cause it to apply a basic set of optimizations. As discussed in Web Aside asm:opt, invoking gcc with flag ‘-O2’ or ‘-O3’ will cause it to apply more extensive optimizations. These can further improve program performance, but they may expand the program size and they may make the program more difficult to debug using standard debugging tools. For our presentation, we will mostly consider code compiled with optimization level 1, even though optimization level 2 has become the accepted standard for most gcc users. We purposely limit the level of optimization to demonstrate how different ways of writing a function in C can affect the efficiency of the code generated by a compiler. We will find that we can write C code that, when compiled just with optimization level 1, vastly outperforms a more naive version compiled with the highest possible optimization levels.
Compilers must be careful to apply only safe optimizations to a program, meaning that the resulting program will have the exact same behavior as would an unoptimized version for all possible cases the program may encounter, up to the limits of the guarantees provided by the C language standards. Constraining the compiler to perform only safe optimizations eliminates possible sources of undesired run-time behavior, but it also means that the programmer must make more of an effort to write programs in a way that the compiler can then transform into efficient machine-level code. To appreciate the challenges of deciding which program transformations are safe or not, consider the following two procedures:

```c
void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2 * *yp;
}
```

At first glance, both procedures seem to have identical behavior. They both add twice the value stored at the location designated by pointer `yp` to that designated by pointer `xp`. On the other hand, function `twiddle2` is more efficient. It requires only three memory references (read `*xp`, read `*yp`, write `*xp`), whereas `twiddle1` requires six (two reads of `*xp`, two reads of `*yp`, and two writes of `*xp`). Hence, if a compiler is given procedure `twiddle1` to compile, one might think it could generate more efficient code based on the computations performed by `twiddle2`.

Consider, however, the case in which `xp` and `yp` are equal. Then function `twiddle1` will perform the following computations:

```c
*xp += *xp; /* Double value at xp */
*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:

```c
*xp += 2* *xp; /* Triple value at xp */
```

The result will be that the value at `xp` will be increased by a factor of 3. The compiler knows nothing about how `twiddle1` will be called, and so it must assume that arguments `xp` and `yp` can be equal. It therefore cannot generate code in the style of `twiddle2` as an optimized version of `twiddle1`.

The case where two pointers may designate the same memory location is known as memory aliasing. In performing only safe optimizations, the compiler...
must assume that different pointers may be aliased. As another example, for a
program with pointer variables \( p \) and \( q \), consider the following code sequence:

\[
x = 1000; \ y = 3000;
*q = y; \quad /* 3000 */
*p = x; \quad /* 1000 */
t1 = *q; \quad /* 1000 or 3000 */
\]

The value computed for \( t1 \) depends on whether or not pointers \( p \) and \( q \) are
aliased—if not, it will equal 3000, but if so it will equal 1000. This leads to one
of the major **optimization blockers**, aspects of programs that can severely limit
the opportunities for a compiler to generate optimized code. If a compiler cannot
determine whether or not two pointers may be aliased, it must assume that either
case is possible, limiting the set of possible optimizations.

**Practice Problem 5.1**

The following problem illustrates the way memory aliasing can cause unexpected
program behavior. Consider the following procedure to swap two values:

```c
/* Swap value x at xp with value y at yp */
void swap(int *xp, int *yp)
{
    *xp = *xp + *yp; /* x+y */
    *yp = *xp - *yp; /* x+y-y=x */
    *xp = *xp - *yp; /* x+y-x=y */
}
```

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, con-
sider the following two procedures:

```c
int f();

int func1() {
    return f() + f() + f() + f();
}

int func2() {
    return 4*f();
}
```

It might seem at first that both compute the same result, but with \( func2 \) calling \( f \)
only once, whereas \( func1 \) calls it four times. It is tempting to generate code in the
style of \( func2 \) when given \( func1 \) as the source.
Consider, however, the following code for \( f \):

```c
1 int counter = 0;
2 int f() {
3     return counter++;
4 }
```

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 \( \text{func1} \) would return \( 0 + 1 + 2 + 3 = 6 \), whereas a call to \( \text{func2} \) would return \( 4 \cdot 0 = 0 \), assuming both started with global variable \( \text{counter} \) 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 \( \text{func2} \). Instead, the compiler assumes the worst case and leaves function calls intact.

---

**Aside** Optimizing function calls by inline substitution

As described in Web Aside *asm:opt*, code involving function calls can be optimized by a process known as *inline substitution* (or simply “inlining”), where the function call is replaced by the code for the body of the function. For example, we can expand the code for \( \text{func1} \) by substituting four instantiations of function \( f \):

```c
/* Result of inlining \( f \) in \( \text{func1} \) */
int func1in() {
    int t = counter++; /* +0 */
    t += counter++; /* +1 */
    t += counter++; /* +2 */
    t += counter++; /* +3 */
    return t;
}
```

This transformation both reduces the overhead of the function calls and allows further optimization of the expanded code. For example, the compiler can consolidate the updates of global variable \( \text{counter} \) in \( \text{func1in} \) to generate an optimized version of the function:

```c
/* Optimization of inlined code */
int func1opt() {
    int t = 4 * counter + 6;
    counter = t + 4;
    return t;
}
```

This code faithfully reproduces the behavior of \( \text{func1} \) for this particular definition of function \( f \).

Recent versions of *gcc* attempt this form of optimization, either when directed to with the command-line option ‘-finline’ or for optimization levels 2 or higher. Since we are considering optimization level 1 in our presentation, we will assume that the compiler does not perform inline substitution.
Among compilers, gcc is considered adequate, but not exceptional, in terms of its optimization capabilities. It performs basic optimizations, but it does not perform the radical transformations on programs that more “aggressive” compilers do. As a consequence, programmers using gcc must put more effort into writing programs in a way that simplifies the compiler’s task of generating efficient code.

5.2 Expressing Program Performance

We introduce the metric cycles per element, abbreviated “CPE,” as a way to express program performance in a way that can guide us in improving the code. CPE measurements help us understand the loop performance of an iterative program at a detailed level. It is appropriate for programs that perform a repetitive computation, such as processing the pixels in an image or computing the elements in a matrix product.

The sequencing of activities by a processor is controlled by a clock providing a regular signal of some frequency, usually expressed in gigahertz (GHz), billions of cycles per second. For example, when product literature characterizes a system as a “4 GHz” processor, it means that the processor clock runs at $4.0 \times 10^9$ cycles per second. The time required for each clock cycle is given by the reciprocal of the clock frequency. These typically are expressed in nanoseconds (1 nanosecond is $10^{-9}$ seconds), or picoseconds (1 picosecond is $10^{-12}$ seconds). For example, the period of a 4 GHz clock can be expressed as either 0.25 nanoseconds or 250 picoseconds. From a programmer’s perspective, it is more instructive to express measurements in clock cycles rather than nanoseconds or picoseconds. That way, the measurements express how many instructions are being executed rather than how fast the clock runs.

Many procedures contain a loop that iterates over a set of elements. For example, functions $p\text{sum}1$ and $p\text{sum}2$ in Figure 5.1 both compute the prefix sum of a vector of length $n$. For a vector $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, the prefix sum $\vec{p} = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ is defined as

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

Function $p\text{sum}1$ computes one element of the result vector per iteration. The second uses a technique known as loop unrolling to compute two elements per iteration. We will explore the benefits of loop unrolling later in this chapter. See Problems 5.11, 5.12, and 5.21 for more about analyzing and optimizing the prefix-sum computation.

The time required by such a procedure can be characterized as a constant plus a factor proportional to the number of elements processed. For example, Figure 5.2 shows a plot of the number of clock cycles required by the two functions for a range of values of $n$. Using a least squares fit, we find that the run times (in clock cycles) for $p\text{sum}1$ and $p\text{sum}2$ can be approximated by the equations $496 + 10.0n$ and $500 + 6.5n$, respectively. These equations indicate an overhead of 496 to 500
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long int n)
{
    long int i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i-1] + a[i];
}

void psum2(float a[], float p[], long int n)
{
    long int i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2) {
        float mid_val = p[i-1] + a[i];
        p[i] = mid_val;
        p[i+1] = mid_val + a[i+1];
    }
    /* For odd n, finish remaining element */
    if (i < n)
        p[i] = p[i-1] + a[i];
}

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

Figure 5.2  Performance of prefix-sum functions. The slope of the lines indicates the number of clock cycles per element (CPE).
cycles due to the timing code and to initiate the procedure, set up the loop, and complete the procedure, plus a linear factor of 6.5 or 10.0 cycles per element. For large values of \( n \) (say, greater than 200), the run times will be dominated by the linear factors. We refer to the coefficients in these terms as the effective number of cycles per element, abbreviated “CPE.” We prefer measuring the number of cycles per element rather than the number of cycles per iteration, because techniques such as loop unrolling allow us to use fewer iterations to complete the computation, but our ultimate concern is how fast the procedure will run for a given vector length. We focus our efforts on minimizing the CPE for our computations. By this measure, \( psum2 \), with a CPE of 6.50, is superior to \( psum1 \), with a CPE of 10.0.

**Aside** What is a least squares fit?

For a set of data points \((x_1, y_1), \ldots, (x_n, y_n)\), we often try to draw a line that best approximates the X-Y trend represented by this data. With a least squares fit, we look for a line of the form \( y = mx + b \) that minimizes the following error measure:

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

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.

**Practice Problem 5.2**

Later in this chapter, we will start with a single function and generate many different variants that preserve the function’s behavior, but with different performance characteristics. For three of these variants, we found that the run times (in clock cycles) can be approximated by the following functions:

- Version 1: \( 60 + 35n \)
- Version 2: \( 136 + 4n \)
- Version 3: \( 157 + 1.25n \)

For what values of \( n \) would each version be the fastest of the three? Remember that \( n \) will always be an integer.

**5.3 Program Example**

To demonstrate how an abstract program can be systematically transformed into more efficient code, we will use a running example based on the vector data structure shown in Figure 5.3. A vector is represented with two blocks of memory: the header and the data array. The header is a structure declared as follows:
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 integer (C int), single-precision floating-point (C float), and double-precision floating-point (C double) data. We do this by compiling and running the program separately for different type declarations, such as the following for data type int:

```c
typedef int data_t;
```

We allocate the data array block to store the vector elements as an array of len objects of type data_t.

Figure 5.4 shows some basic procedures for generating vectors, accessing vector elements, and determining the length of a vector. An important feature to note is that get_vec_element, the vector access routine, performs bounds checking for every vector reference. This code is similar to the array representations used in many other languages, including Java. Bounds checking reduces the chances of program error, but it can also slow down program execution.

As an optimization example, consider the code shown in Figure 5.5, which combines all of the elements in a vector into a single value according to some operation. By using different definitions of compile-time constants IDENT and OP, the code can be recompiled to perform different operations on the data. In particular, using the declarations

```c
#define IDENT 0
#define OP +
```

it sums the elements of the vector. Using the declarations

```c
#define IDENT 1
#define OP *
```

it computes the product of the vector elements.

In our presentation, we will proceed through a series of transformations of the code, writing different versions of the combining function. To gauge progress,
/* Create vector of specified length */
vec_ptr new_vec(long int len)
{
    /* Allocate header structure */
    vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
    if (!result)
        return NULL; /* Couldn't allocate storage */
    result->len = len;
    /* Allocate array */
    if (len > 0) {
        data_t *data = (data_t *)calloc(len, sizeof(data_t));
        if (!data) {
            free((void *) result);
            return NULL; /* Couldn't allocate storage */
        }
        result->data = data;
    } else
        result->data = NULL;
    return result;
}

/* Retrieve vector element and store at dest. */
/* Return 0 (out of bounds) or 1 (successful) */
int get_vec_element(vec_ptr v, long int index, data_t *dest)
{
    if (index < 0 || index >= v->len)
        return 0;
    *dest = v->data[index];
    return 1;
}

/* Return length of vector */
long int vec_length(vec_ptr v)
{
    return v->len;
}

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.
/* Implementation with maximum use of data abstraction */
void combine1(vec_ptr v, data_t *dest)
{
  long int i;
  *dest = IDENT;
  for (i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}

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

we will measure the CPE performance of the functions on a machine with an Intel Core i7 processor, which we will refer to as our reference machine. Some characteristics of this processor were given in Section 3.1. These measurements characterize performance in terms of how the programs run on just one particular machine, and so there is no guarantee of comparable performance on other combinations of machine and compiler. However, we have compared the results with those for a number of different compiler/processor combinations and found them quite comparable.

As we proceed through a set of transformations, we will find that many lead to only minimal performance gains, while others have more dramatic effects. Determining which combinations of transformations to apply is indeed part of the “black art” of writing fast code. Some combinations that do not provide measurable benefits are indeed ineffective, while others are important as ways to enable further optimizations by the compiler. In our experience, the best approach involves a combination of experimentation and analysis: repeatedly attempting different approaches, performing measurements, and examining the assembly-code representations to identify underlying performance bottlenecks.

As a starting point, the following are CPE measurements for combine1 running on our reference machine, trying all combinations of data type and combining operation. For single-precision and double-precision floating-point data, our experiments on this machine gave identical performance for addition, but differing performance for multiplication. We therefore report five CPE values: integer addition and multiplication, floating-point addition, single-precision multiplication (labeled “F *”), and double-precision multiplication (labeled “D *”).

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>combine1</td>
<td>485</td>
<td>Abstract unoptimized</td>
<td>29.02</td>
<td>29.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>combine1</td>
<td>485</td>
<td>Abstract -01</td>
<td>12.00</td>
<td>12.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can see that our measurements are somewhat imprecise. The more likely CPE number for integer sum and product is 29.00, rather than 29.02 or 29.21. Rather than “fudging” our numbers to make them look good, we will present the measurements we actually obtained. There are many factors that complicate the task of reliably measuring the precise number of clock cycles required by some code sequence. It helps when examining these numbers to mentally round the results up or down by a few hundredths of a clock cycle.

The unoptimized code provides a direct translation of the C code into machine code, often with obvious inefficiencies. By simply giving the command-line option ‘-O1’, we enable a basic set of optimizations. As can be seen, this significantly improves the program performance—more than a factor of two—with no effort on behalf of the programmer. In general, it is good to get into the habit of enabling at least this level of optimization. For the remainder of our measurements, we use optimization levels 1 and higher in generating and measuring our programs.

5.4 Eliminating Loop Inefficiencies

Observe that procedure combine1, as shown in Figure 5.5, calls function vec_length as the test condition of the for loop. Recall from our discussion of how to translate code containing loops into machine-level programs (Section 3.6.5) that the test condition must be evaluated on every iteration of the loop. On the other hand, the length of the vector does not change as the loop proceeds. We could therefore compute the vector length only once and use this value in our test condition.

Figure 5.6 shows a modified version called combine2, which calls vec_length at the beginning and assigns the result to a local variable length. This transformation has noticeable effect on the overall performance for some data types and

```c
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

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.
operations, and minimal or even none for others. In any case, this transformation is required to eliminate inefficiencies that would become bottlenecks as we attempt further optimizations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>F *</td>
</tr>
<tr>
<td>combine1</td>
<td>485</td>
<td>Abstract -01</td>
<td>12.00</td>
<td>12.01</td>
</tr>
<tr>
<td>combine2</td>
<td>486</td>
<td>Move vec_length</td>
<td>8.03</td>
<td>10.09</td>
</tr>
</tbody>
</table>

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. To improve the code, the programmer must often help the compiler by explicitly performing code motion.

As an extreme example of the loop inefficiency seen in combine1, consider the procedure lower1 shown in Figure 5.7. This procedure is styled after routines submitted by several students as part of a network programming project. Its purpose is to convert all of the uppercase letters in a string to lowercase. The procedure steps through the string, converting each uppercase character to lowercase. The case conversion involves shifting characters in the range ‘A’ to ‘Z’ to the range ‘a’ to ‘z’.

The library function strlen is called as part of the loop test of lower1. Although strlen is typically implemented with special x86 string-processing instructions, its overall execution is similar to the simple version that is also shown in Figure 5.7. Since strings in C are null-terminated character sequences, strlen can only determine the length of a string by stepping through the sequence until it hits a null character. For a string of length $n$, strlen takes time proportional to $n$. Since strlen is called in each of the $n$ iterations of lower1, the overall run time of lower1 is quadratic in the string length, proportional to $n^2$.

This analysis is confirmed by actual measurements of the functions for different length strings, as shown in Figure 5.8 (and using the library version of strlen). The graph of the run time for lower1 rises steeply as the string length increases (Figure 5.8(a)). Figure 5.8(b) shows the run times for seven different lengths (not the same as shown in the graph), each of which is a power of 2. Observe that for lower1 each doubling of the string length causes a quadrupling of the run time. This is a clear indicator of a quadratic run time. For a string of length 1,048,576, lower1 requires over 13 minutes of CPU time.
Figure 5.7  Lowercase conversion routines. The two procedures have radically different performance.

Function lower2 shown in Figure 5.7 is identical to that of lower1, except that we have moved the call to strlen out of the loop. The performance improves dramatically. For a string length of 1,048,576, the function requires just 1.5 milliseconds—over 500,000 times faster than lower1. Each doubling of the string length causes a doubling of the run time—a clear indicator of linear run time. For longer strings, the run-time improvement will be even greater.

In an ideal world, a compiler would recognize that each call to strlen in the loop test will return the same result, and thus the call could be moved out of the loop. This would require a very sophisticated analysis, since strlen checks
Figure 5.8  **Comparative performance of lowercase conversion routines.** The original code `lower1` has a quadratic run time due to an inefficient loop structure. The modified code `lower2` has a linear run time.

When the elements of the string and these values are changing as `lower1` proceeds, the compiler would need to detect that even though the characters within the string are changing, none are being set from nonzero to zero, or vice versa. Such an analysis is well beyond the ability of even the most sophisticated compilers, even if they employ inlining, and so programmers must do such transformations themselves.

This example illustrates a common problem in writing programs, in which a seemingly trivial piece of code has a hidden asymptotic inefficiency. One would not expect a lowercase conversion routine to be a limiting factor in a program’s performance. Typically, programs are tested and analyzed on small data sets, for which the performance of `lower1` is adequate. When the program is ultimately deployed, however, it is entirely possible that the procedure could be applied to strings of over one million characters. All of a sudden this benign piece of code has become a major performance bottleneck. By contrast, the performance of `lower2` will be adequate for strings of arbitrary length. Stories abound of major programming projects in which problems of this sort occur. Part of the job of a competent programmer is to avoid ever introducing such asymptotic inefficiency.
**Practice Problem 5.3**

Consider the following functions:

```c
int min(int x, int y) { return x < y ? x : y; }
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int square(int x) { return x*x; }
```

The following three code fragments call these functions:

A. ```c
for (i = min(x, y); i < max(x, y); incr(&i, 1))
   t += square(i);
```

B. ```c
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
   t += square(i);
```

C. ```c
int low = min(x, y);
int high = max(x, y);
for (i = low; i < high; incr(&i, 1))
   t += square(i);
```

Assume `x` equals 10 and `y` equals 100. Fill in the following table indicating the number of times each of the four functions is called in code fragments A–C:

<table>
<thead>
<tr>
<th>Code</th>
<th>min</th>
<th>max</th>
<th>incr</th>
<th>square</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5.5 Reducing Procedure Calls

As we have seen, procedure calls can incur overhead and also block most forms of program optimization. We can see in the code for `combine2` (Figure 5.6) that `get_vec_element` is called on every loop iteration to retrieve the next vector element. This function checks the vector index `i` against the loop bounds with every vector reference, a clear source of inefficiency. Bounds checking might be a useful feature when dealing with arbitrary array accesses, but a simple analysis of the code for `combine2` shows that all references will be valid.

Suppose instead that we add a function `get_vec_start` to our abstract data type. This function returns the starting address of the data array, as shown in Figure 5.9. We could then write the procedure shown as `combine3` in this figure, having no function calls in the inner loop. Rather than making a function call to retrieve each vector element, it accesses the array directly. A purist might say that this transformation seriously impairs the program modularity. In principle, the user of the vector abstract data type should not even need to know that the vector...
Section 5.6 Eliminating Unneeded Memory References

The code for combine3 accumulates the value being computed by the combining operation at the location designated by the pointer dest. This attribute can be seen by examining the assembly code generated for the compiled loop. We show

---

```c
data_t *get_vec_start(vec_ptr v)
{
    return v->data;
}
```

---

```c
/* Direct access to vector data */
void combine3(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

---

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

The resulting improvement is surprisingly modest, only improving the performance for integer sum. Again, however, this inefficiency would become a bottleneck as we attempt further optimizations. We will return to this function later (Section 5.11.2) and see why the repeated bounds checking by combine2 does not make its performance much worse. For applications in which performance is a significant issue, one must often compromise modularity and abstraction for speed. It is wise to include documentation on the transformations applied, as well as the assumptions that led to them, in case the code needs to be modified later.

### 5.6 Eliminating Unneeded Memory References

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>combine2</td>
<td>486</td>
<td>Move vec_length</td>
<td>8.03</td>
<td>8.09</td>
</tr>
<tr>
<td>combine3</td>
<td>491</td>
<td>Direct data access</td>
<td>6.01</td>
<td>8.01</td>
</tr>
</tbody>
</table>

Contents are stored as an array, rather than as some other data structure such as a linked list. A more pragmatic programmer would argue that this transformation is a necessary step toward achieving high-performance results.
here the x86-64 code generated for data type float and with multiplication as the combining operation:

```
combine3: data_t = float, OP = *
i in %rdx, data in %rax, dest in %rbp
1 .L498:           loop:
2   movss (%rbp), %xmm0  Read product from dest
3   mulss (%rax,%rdx,4), %xmm0  Multiply product by data[i]
4   movss %xmm0, (%rbp)  Store product at dest
5   addq $1, %rdx  Increment i
6   cmpq %rdx, %r12  Compare i:limit
7   jg .L498  If >, goto loop
```

**Aside**  Understanding x86-64 floating-point code

We cover floating-point code for x86-64, the 64-bit version of the Intel instruction set in Web Aside asm:sse, but the program examples we show in this chapter can readily be understood by anyone familiar with IA32 code. Here, we briefly review the relevant aspects of x86-64 and its floating-point instructions.

The x86-64 instruction set extends the 32-bit registers of IA32, such as %eax, %edi, and %esp, to 64-bit versions, with ‘r’ replacing ‘e’, e.g., %rax, %rdi, and %rsp. Eight more registers are available, named %r8-%r15, greatly improving the ability to hold temporary values in registers. Suffix ‘q’ is used on integer instructions (e.g., addq, cmpq) to indicate 64-bit operations.

Floating-point data are held in a set of XMM registers, named %xmm0-%xmm15. Each of these registers is 128 bits long, able to hold four single-precision (float) or two double-precision (double) floating-point numbers. For our initial presentation, we will only make use of instructions that operate on single values held in SSE registers.

The movss instruction copies one single-precision number. Like the various mov instructions of IA32, both the source and the destination can be memory locations or registers, but it uses XMM registers, rather than general-purpose registers. The mulss instruction multiplies single-precision numbers, updating its second operand with the product. Again, the source and destination operands can be memory locations or XMM registers.

We see in this loop code that the address corresponding to pointer dest is held in register %rbp (unlike in IA32, where %ebp has special use as a frame pointer, its 64-bit counterpart %rbp can be used to hold arbitrary data). On iteration i, the program reads the value at this location, multiplies it by data[i], and stores the result back at dest. This reading and writing is wasteful, since the value read from dest at the beginning of each iteration should simply be the value written at the end of the previous iteration.

We can eliminate this needless reading and writing of memory by rewriting the code in the style of combine4 in Figure 5.10. We introduce a temporary variable acc that is used in the loop to accumulate the computed value. The result is stored at dest only after the loop has been completed. As the assembly code that follows shows, the compiler can now use register %xmm0 to hold the accumulated value.
/* Accumulate result in local variable */
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}

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

Compared to the loop in combine3, we have reduced the memory operations per iteration from two reads and one write to just a single read.

combine4: data_t = float, OP = *
i in %rdx, data in %rax, limit in %rbp, acc in %xmm0
.L488:
.loop:
    mulss (%rax,%rdx,4), %xmm0 Multiply acc by data[i]
    addq $1, %rdx Increment i
    cmpq %rdx, %rbp Compare limit:i
    jg .L488 If >, goto loop

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>F</td>
</tr>
<tr>
<td>combine3</td>
<td>491</td>
<td>Direct data access</td>
<td>6.01</td>
<td>10.01</td>
</tr>
<tr>
<td>combine4</td>
<td>493</td>
<td>Accumulate in temporary</td>
<td>2.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>

All of our times improve by at least a factor of $2.4 \times$, with the integer addition case dropping to just two clock cycles per element.

Aside Expressing relative performance

The best way to express a performance improvement is as a ratio of the form $T_{\text{old}}/T_{\text{new}}$, where $T_{\text{old}}$ is the time required for the original version and $T_{\text{new}}$ 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 ‘$\times$’ to indicate such a ratio, where the factor “$2.4 \times$” is expressed verbally as “$2.4$ 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 $100 \cdot \frac{T_{\text{old}} - T_{\text{new}}}{T_{\text{new}}}$ or possibly $100 \cdot \frac{T_{\text{old}} - T_{\text{new}}}{T_{\text{old}}}$, or something else? In addition, it is less instructive for large changes. Saying that “performance improved by 140%” is more difficult to comprehend than simply saying that the performance improved by a factor of 2.4.

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Initial</th>
<th>Before loop</th>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine3</td>
<td>[2, 3, 5]</td>
<td>[2, 3, 1]</td>
<td>[2, 3]</td>
<td>[2, 3]</td>
<td>[2, 2]</td>
<td>[2, 36]</td>
</tr>
<tr>
<td>combine4</td>
<td>[2, 3, 5]</td>
<td>[2, 3, 5]</td>
<td>[2, 3]</td>
<td>[2, 3]</td>
<td>[2, 3]</td>
<td>[2, 30]</td>
</tr>
</tbody>
</table>

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

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

**Practice Problem 5.4**

When we use `gcc` to compile `combine3` with command-line option ‘-O2’, we get code with substantially better CPE performance than with ‘-O1':
We achieve performance comparable to that for combine4, except for the case of integer sum, but even it improves significantly. On examining the assembly code generated by the compiler, we find an interesting variant for the inner loop:

```
combine3: data_t = float, OP = *, compiled -O2
i in %rdx, data in %rax, limit in %rbp, dest at %rx12
Product in %xmm0

.L560:
loop:
2  muls (%rax,%rdx,4), %xmm0  Multiply product by data[i]
3  addq $1, %rdx  Increment i
4  cmpq %rdx, %rbp  Compare limit:i
5  movss %xmm0, (%r12)  Store product at dest
6  jg .L560  If >, goto loop
```

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

```
combine3: data_t = float, OP = *, compiled -O1
i in %rdx, data in %rax, dest in %rbp

.L498:
loop:
2  movss (%rbp), %xmm0  Read product from dest
3  muls (%rax,%rdx,4), %xmm0  Multiply product by data[i]
4  movss %xmm0, (%rbp)  Store product at dest
5  addq $1, %rdx  Increment i
6  cmpq %rdx, %r12  Compare i:limit
7  jg .L498  If >, goto loop
```

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

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

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

C. Explain either why this optimization preserves the desired behavior, or give an example where it would produce different results than the less optimized code.
With this final transformation, we reached a point where we require just 2–5 clock cycles for each element to be computed. This is a considerable improvement over the original 11–13 cycles when we first enabled optimization. We would now like to see just what factors are constraining the performance of our code and how we can improve things even further.

5.7 Understanding Modern Processors

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

To understand ways to improve performance, we require a basic understanding of the microarchitectures of modern processors. Due to the large number of transistors that can be integrated onto a single chip, modern microprocessors employ complex hardware that attempts to maximize program performance. One result is that their actual operation is far different from the view that is perceived by looking at machine-level programs. At the code level, it appears as if instructions are executed one at a time, where each instruction involves fetching values from registers or memory, performing an operation, and storing results back to a register or memory location. In the actual processor, a number of instructions are evaluated simultaneously, a phenomenon referred to as instruction-level parallelism. In some designs, there can be 100 or more instructions “in flight.” Elaborate mechanisms are employed to make sure the behavior of this parallel execution exactly captures the sequential semantic model required by the machine-level program. This is one of the remarkable feats of modern microprocessors: they employ complex and exotic microarchitectures, in which multiple instructions can be executed in parallel, while presenting an operational view of simple sequential instruction execution.

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

### 5.7.1 Overall Operation

Figure 5.11 shows a very simplified view of a modern microprocessor. Our hypothetical processor design is based loosely on the structure of the Intel Core i7 processor design, which is often referred to by its project code name “Nehalem” [99]. The Nehalem 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 machine-level program. The overall design has two main parts: the **instruction control unit** (ICU), which is responsible for reading a sequence of instructions from memory and generating from these a set of primitive operations to perform on program data, and the **execution unit** (EU), which then executes these operations. Compared to the simple **in-order** pipeline we studied in Chapter 4, out-of-order processors require far greater and more
complex hardware, but they are better at achieving higher degrees of instruction-level parallelism.

The ICU reads the instructions from an instruction cache—a special high-speed memory containing the most recently accessed instructions. In general, the ICU fetches well ahead of the currently executing instructions, so that it has enough time to decode these and send operations down to the EU. One problem, however, is that when a program hits a branch,\(^1\) there are two possible directions the program might go. The branch can be taken, with control passing to the branch target. Alternatively, the branch can be not taken, with control passing to the next instruction in the instruction sequence. Modern processors employ a technique known as branch prediction, in which they guess whether or not a branch will be taken and also predict the target address for the branch. Using a technique known as speculative execution, the processor begins fetching and decoding instructions at where it predicts the branch will go, and even begins executing these operations before it has been determined whether or not the branch prediction was correct. If it later determines that the branch was predicted incorrectly, it resets the state to that at the branch point and begins fetching and executing instructions in the other direction. The block labeled “Fetch control” incorporates branch prediction to perform the task of determining which instructions to fetch.

The instruction decoding logic takes the actual program instructions and converts them into a set of primitive operations (sometimes referred to as micro-operations). Each of these operations performs some simple computational task such as adding two numbers, reading data from memory, or writing data to memory. For machines with complex instructions, such as x86 processors, an instruction can be decoded into a variable number of operations. The details of how instructions are decoded into sequences of more primitive operations varies between machines, and this information is considered highly proprietary. Fortunately, we can optimize our programs without knowing the low-level details of a particular machine implementation.

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

\[
\text{addl}\ %\text{eax},%\text{edx}
\]

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

\[
\text{addl}\ %\text{eax},4(%\text{edx})
\]

yields multiple operations, separating the memory references from the arithmetic operations. This particular instruction would be decoded as three operations: one to load a value from memory into the processor, one to add the loaded value to the

---

\(^1\) 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.
value in register `%eax`, and one to \textit{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.

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 \textit{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, based on those of the Intel Core i7. We can see that three functional units are dedicated to computation, while the remaining two are for reading (load) and writing (store) memory. Each computational unit can perform multiple different operations: all can perform at least basic integer operations, such as addition and bit-wise logical operations. Floating-point operations and integer multiplication require more complex hardware, and so these can only be handled by specific functional units.

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

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

Within the ICU, the \textit{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 \textit{register file} containing the integer, floating-point, and more recently SSE registers as part of the retirement unit, because this unit controls the updating of these registers. As an instruction is decoded, information about it is placed into a first-in, first-out queue. This information remains in the queue until one of two outcomes occurs. First, once the operations for the instruction have completed and any branch points leading to this instruction are confirmed as having been correctly predicted, the instruction can be \textit{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 \textit{flushed}, discarding any results that may have been computed. By this means, mispredictions will not alter the program state.
As we have described, any updates to the program registers occur only as instructions are being retired, and this takes place only after the processor can be certain that any branches leading to this instruction have been correctly predicted. To expedite the communication of results from one instruction to another, much of this information is exchanged among the execution units, shown in the figure as “Operation results.” As the arrows in the figure show, the execution units can send results directly to each other. This is a more elaborate form of the data forwarding techniques we incorporated into our simple processor design in Section 4.5.7.

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

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

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

IBM first implemented out-of-order processing with the IBM 360/91 processor in 1966, but just to execute the floating-point instructions. For around 25 years, out-of-order processing was considered an exotic technology, found only in machines striving for the highest possible performance, until IBM reintroduced it in the RS/6000 line of workstations in 1990. This design became the basis for the IBM/Motorola PowerPC line, with the model 601, introduced in 1993, becoming the first single-chip microprocessor to use out-of-order processing. Intel introduced out-of-order processing with its PentiumPro model in 1995, with an underlying microarchitecture similar to that of the Core i7.

### 5.7.2 Functional Unit Performance

Figure 5.12 documents the performance of some of the arithmetic operations for an Intel Core i7, determined by both measurements and by reference to Intel liter-
Section 5.7 Understanding Modern Processors

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer</th>
<th>Single-precision</th>
<th>Double-precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Issue</td>
<td>Latency</td>
</tr>
<tr>
<td>Addition</td>
<td>1</td>
<td>0.33</td>
<td>3</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Division</td>
<td>11–21</td>
<td>5–13</td>
<td>10–15</td>
</tr>
</tbody>
</table>

Figure 5.12 Latency and issue time characteristics of Intel Core i7 arithmetic operations. Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two operations. The times for division depend on the data values.

These timings are typical for other processors as well. Each operation is characterized by its latency, meaning the total time required to perform the operation, and the issue time, meaning the minimum number of clock cycles between two successive operations of the same type.

We see that the latencies increase as the word sizes increase (e.g., from single to double precision), for more complex data types (e.g., from integer to floating point), and for more complex operations (e.g., from addition to multiplication).

We see also that most forms of addition and multiplication operations have issue times of 1, meaning that on each clock cycle, the processor can start a new one of these operations. This short issue time is achieved through the use of pipelining. A pipelined function unit is implemented as a series of stages, each of which performs part of the operation. For example, a typical floating-point adder contains three stages (and hence the three-cycle latency): one to process the exponent values, one to add the fractions, and one to round the result. The arithmetic operations can proceed through the stages in close succession rather than waiting for one operation to complete before the next begins. This capability can be exploited only if there are successive, logically independent operations to be performed. Functional units with issue times of 1 cycle are said to be fully pipelined: they can start a new operation every clock cycle. The issue time of 0.33 given for integer addition is due to the fact that the hardware has three fully pipelined functional units capable of performing integer addition. The processor has the potential to perform three additions every clock cycle. We see also that the divider (used for integer and floating-point division, as well as floating-point square root) is not fully pipelined—its issue time is just a few cycles less than its latency. What this means is that the divider must complete all but the last few steps of a division before it can begin a new one. We also see the latencies and issue times for division are given as ranges, because some combinations of dividend and divisor require more steps than others. The long latency and issue times of division make it a comparatively costly operation.

A more common way of expressing issue time is to specify the maximum throughput of the unit, defined as the reciprocal of the issue time. A fully pipelined functional unit has a maximum throughput of one operation per clock cycle, while units with higher issue times have lower maximum throughput.
Circuit designers can create functional units with wide ranges of performance characteristics. Creating a unit with short latency or with pipelining requires more hardware, especially for more complex functions such as multiplication and floating-point operations. Since there is only a limited amount of space for these units on the microprocessor chip, CPU designers must carefully balance the number of functional units and their individual performance to achieve optimal overall performance. They evaluate many different benchmark programs and dedicate the most resources to the most critical operations. As Figure 5.12 indicates, integer multiplication and floating-point multiplication and addition were considered important operations in design of the Core i7, even though a significant amount of hardware is required to achieve the low latencies and high degree of pipelining shown. On the other hand, division is relatively infrequent and difficult to implement with either short latency or full pipelining.

Both the latencies and the issue times (or equivalently, the maximum throughput) of these arithmetic operations can affect the performance of our combining functions. We can express these effects in terms of two fundamental bounds on the CPE values:

<table>
<thead>
<tr>
<th>Bound</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>F * D *</td>
</tr>
<tr>
<td>Latency</td>
<td>1.00</td>
<td>3.00 3.00 4.00 5.00</td>
</tr>
<tr>
<td>Throughput</td>
<td>1.00</td>
<td>1.00 1.00 1.00</td>
</tr>
</tbody>
</table>

The latency bound gives a minimum value for the CPE for any function that must perform the combining operation in a strict sequence. The throughput bound gives a minimum bound for the CPE based on the maximum rate at which the functional units can produce results. For example, since there is only one multiplier, and it has an issue time of 1 clock cycle, the processor cannot possibly sustain a rate of more than one multiplication per clock cycle. We noted earlier that the processor has three functional units capable of performing integer addition, and so we listed the issue time for this operation as 0.33. Unfortunately, the need to read elements from memory creates an additional throughput bound for the CPE of 1.00 for the combining functions. We will demonstrate the effect of both of the latency and throughput bounds with different versions of the combining functions.

### 5.7.3 An Abstract Model of Processor Operation

As a tool for analyzing the performance of a machine-level program executing on a modern processor, we will use a data-flow representation of programs, a graphical notation showing how the data dependencies between the different operations constrain the order in which they are executed. These constraints then lead to critical paths in the graph, putting a lower bound on the number of clock cycles required to execute a set of machine instructions.
Before proceeding with the technical details, it is instructive to examine the CPE measurements obtained for function combine4, our fastest code up to this point:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>++</td>
<td>+ F D</td>
</tr>
<tr>
<td>combine4</td>
<td>493</td>
<td>Accumulate in temporary</td>
<td>2.00</td>
<td>3.00 4.00 5.00</td>
</tr>
<tr>
<td>Latency bound</td>
<td></td>
<td></td>
<td>1.00</td>
<td>3.00 4.00 5.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00 1.00 1.00</td>
</tr>
</tbody>
</table>

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

From Machine-Level Code to Data-Flow Graphs

Our data-flow representation of programs is informal. We only want to use it as a way to visualize how the data dependencies in a program dictate its performance. We present the data-flow notation by working with combine4 (Figure 5.10, page 493) as an example. We focus just on the computation performed by the loop, since this is the dominating factor in performance for large vectors. We consider the case of floating-point data with multiplication as the combining operation, although other combinations of data type and operation have nearly identical structure. The compiled code for this loop consists of four instructions, with registers %rdx holding loop index \( i \), %rax holding array address data, %rcx holding loop bound limit, and %xmm0 holding accumulator value acc.

```
combine4: data_t = float, OP = *
i in %rdx, data in %rax, limit in %rbp, acc in %xmm0
.L488:
loop:
mulss (%rax,%rdx,4), %xmm0 Multiply acc by data[i]
addq $1, %rdx Increment i
 cmpq %rdx, %rbp Compare limit:i
  jg .L488 If >, goto loop
```

As Figure 5.13 indicates, with our hypothetical processor design, the four instructions are expanded by the instruction decoder into a series of five operations, with the initial multiplication instruction being expanded into a load operation to read the source operand from memory, and a mul operation to perform the multiplication.
As a step toward generating a data-flow graph representation of the program, the boxes and lines along the left-hand side of Figure 5.13 show how the registers are used and updated by the different operations, with the boxes along the top representing the register values at the beginning of the loop, and those along the bottom representing the values at the end. For example, register `%rax` is only used as a source value by the `load` operation in performing its address calculation, and so the register has the same value at the end of the loop as at the beginning. Similarly, register `%rcx` is only used by the `cmp` operation. Register `%rdx`, on the other hand, is both used and updated within the loop. Its initial value is used by the `load` and `add` operations; its new value is generated by the `add` operation, which is then used by the `cmp` operation. Register `%xmm0` is also updated within the loop by the `mul` operation, which first uses the initial value as a source value.

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

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

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

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

**Local:** These are updated and used within the loop, but there is no dependency from one iteration to another. The condition code registers are examples
Figure 5.14

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

for this loop: they are updated by the cmp operation and used by the jl operation, but this dependency is contained within individual iterations.

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

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

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

In Figure 5.14(a), we also color operators white if they are not part of some chain of dependencies between loop registers. For this example, the compare (cmp) and branch (jl) operations do not directly affect the flow of data in the program. We assume that the Instruction Control Unit predicts that branch will be taken, and hence the program will continue looping. The purpose of the compare and branch operations is to test the branch condition and notify the ICU if it is not. We assume this checking can be done quickly enough that it does not slow down the processor.

In Figure 5.14(b), we have eliminated the operators that were colored white on the left, and we have retained only the loop registers. What we have left is an abstract template showing the data dependencies that form among loop registers due to one iteration of the loop. We can see in this diagram that there are two data dependencies from one iteration to the next. Along one side, we see the dependencies between successive values of program value acc, stored in register %xmm0. The loop computes a new value for acc by multiplying the old value by
Figure 5.15
Data-flow representation of computation by \( n \) iterations by the inner loop of \( \text{combine} \). The sequence of multiplication operations forms a critical path that limits program performance.

Figure 5.15 demonstrates why we achieved a CPE equal to the latency bound of 4 cycles for \( \text{combine} \), when performing single-precision floating-point multiplication. When executing the function, the floating-point multiplier becomes the limiting resource. The other operations required during the loop—manipulating a data element, generated by the load operation. Along the other side, we see the dependencies between successive values of loop index \( i \). On each iteration, the old value is used to compute the address for the load operation, and it is also incremented by the add operation to compute the new value.
and testing loop index i, computing the address of the next data elements, and reading data from memory—proceed in parallel with the multiplier. As each successive value of acc is computed, it is fed back around to compute the next value, but this will not be completed until four cycles later.

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

Other Performance Factors

For the case of integer addition, on the other hand, our measurements of combine4 show a CPE of 2.00, slower than the CPE of 1.00 we would predict based on the chains of dependencies formed along either the left- or the right-hand side of the graph of Figure 5.15. This illustrates the principle that the critical paths in a data-flow representation provide only a lower bound on how many cycles a program will require. Other factors can also limit performance, including the total number of functional units available and the number of data values that can be passed among the functional units on any given step. For the case of integer addition as the combining operation, the data operation is sufficiently fast that the rest of the operations cannot supply data fast enough. Determining exactly why the program requires 2.00 cycles per element would require a much more detailed knowledge of the hardware design than is publicly available.

To summarize our performance analysis of combine4: our abstract data-flow representation of program operation showed that combine4 has a critical path of length \( L \cdot n \) caused by the successive updating of program value acc, and this path limits the CPE to at least \( L \). This is indeed the CPE we measure for all cases except integer addition, which has a measured CPE of 2.00 rather than the CPE of 1.00 we would expect from the critical path length.

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

Practice Problem 5.5

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

\[
a_0 + a_1x + a_2x^2 + \cdots + a_nx^n
\]

This evaluation can be implemented by the following function, having as arguments an array of coefficients \( a \), a value \( x \), and the polynomial degree, \( degree \)
Chapter 5 Optimizing Program Performance

(the value \( n \) in Equation 5.2). In this function, we compute both the successive terms of the equation and the successive powers of \( x \) within a single loop:

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

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

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

Practice Problem 5.6

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

\[
a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n) \cdots))
\]  

(5.3)

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

```c
1 */ Apply Horner's method */
2 double polyh(double a[], double x, int degree)
3 {
4     long int i;
5     double result = a[degree];
6     for (i = degree-1; i >= 0; i--)
7         result = a[i] + x*result;
8     return result;
9 }
```
A. For degree \( n \), how many additions and how many multiplications does this code perform?

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

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

### 5.8 Loop Unrolling

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

Figure 5.16 shows a version of our combining code using two-way loop unrolling. The first loop steps through the array two elements at a time. That is, the loop index \( i \) is incremented by 2 on each iteration, and the combining operation is applied to array elements \( i \) and \( i + 1 \) in a single iteration.

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

We can generalize this idea to unroll a loop by any factor \( k \). To do so, we set the upper limit to be \( n - k + 1 \), and within the loop apply the combining operation to elements \( i \) through \( i + k - 1 \). Loop index \( i \) is incremented by \( k \) in each iteration. The maximum array index \( i + k - 1 \) will then be less than \( n \). We include the second loop to step through the final few elements of the vector one at a time. The body of this loop will be executed between 0 and \( k - 1 \) times. For \( k = 2 \), we could use a simple conditional statement to optionally add a final iteration, as we did with the function \( psum2 \) (Figure 5.1). For \( k > 2 \), the finishing cases are better expressed with a loop, and so we adopt this programming convention for \( k = 2 \) as well.
/* Unroll loop by 2 */
void combine5(vec_ptr v, data_t *dest)
{
  long int i;
  long int length = vec_length(v);
  long int limit = length-1;
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;

  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    acc = (acc OP data[i]) OP data[i+1];
  }

  /* Finish any remaining elements */
  for (; i < length; i++) {
    acc = acc OP data[i];
  }
  *dest = acc;
}

Figure 5.16 Unrolling loop by factor $k = 2$. Loop unrolling can reduce the effect of loop overhead.

Practice Problem 5.7

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>combine4</td>
<td>493</td>
<td>No unrolling</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>combine5</td>
<td>510</td>
<td>Unroll by $\times 2$</td>
<td>2.00</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unroll by $\times 3$</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Latency bound</td>
<td></td>
<td></td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

We see that CPEs for both integer addition and multiplication improve, while those for the floating-point operations do not. Figure 5.17 shows CPE measurements when unrolling the loop by up to a factor of 6. We see that the trends we
observed for unrolling by 2 and 3 continue—it does not help the floating-point operations, while both integer addition and multiplication drop down to CPEs of 1.00. Several phenomena contribute to these measured values of CPE. For the case of integer addition, we see that unrolling by a factor of 2 makes no difference, but unrolling by a factor of 3 drops the CPE to 1.00, achieving both the latency and the throughput bounds for this operation. This result can be attributed to the benefits of reducing loop overhead operations. By reducing the number of overhead operations relative to the number of additions required to compute the vector sum, we can reach the point where the one-cycle latency of integer addition becomes the performance-limiting factor.

The improving CPE for integer multiplication is surprising. We see that for unrolling factor $k$ between 1 and 3, the CPE is $3.00/k$. It turns out that the compiler is making an optimization based on a reassociation transformation, altering the order in which values are combined. We will cover this transformation in Section 5.9.2. The fact that gcc applies this transformation to integer multiplication but not to floating-point addition or multiplication is due to the associativity properties of the different operations and data types, as will also be discussed later.

To understand why the three floating-point cases do not improve by loop unrolling, consider the graphical representation for the inner loop, shown in Figure 5.18 for the case of single-precision multiplication. We see here that the `mulss` instructions each get translated into two operations: one to load an array element from memory, and one to multiply this value by the accumulated value. We see here that register `%xmm0` gets read and written twice in each execution of the loop. We can rearrange, simplify, and abstract this graph, following the process shown in Figure 5.19 to obtain the template shown in Figure 5.19(b). We then replicate this template $n/2$ times to show the computation for a vector of length $n$, obtaining the data-flow representation shown in Figure 5.20. We see here that there is still a critical path of $n$ `mul` operations in this graph—there are half as many iterations, but each iteration has two multiplication operations in sequence. Since the critical path was the limiting factor for the performance of the code without loop unrolling, it remains so with simple loop unrolling.
Figure 5.18
Graphical representation of inner-loop code for combine5. Each iteration has two mulss instructions, each of which is translated into a load and a mul operation.

Each iteration has two mulss instructions, each of which is translated into a load and a mul operation.

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

Aside
Getting the compiler to unroll loops

Loop unrolling can easily be performed by a compiler. Many compilers do it routinely whenever the optimization level is set sufficiently high. gcc will perform loop unrolling when invoked with command-line option ‘-funroll-loops’.
5.9 Enhancing Parallelism

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

### 5.9.1 Multiple Accumulators

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>$\times$</td>
</tr>
<tr>
<td>combine4</td>
<td>493</td>
<td>Accumulate in temporary</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>combine5</td>
<td>510</td>
<td>Unroll by $\times 2$</td>
<td>2.00</td>
<td>1.50</td>
</tr>
<tr>
<td>combine6</td>
<td>515</td>
<td>Unroll $\times 2$, parallelism $\times 2$</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td>Latency bound</td>
<td></td>
<td></td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Figure 5.22 demonstrates the effect of applying this transformation to achieve $k$-way loop unrolling and $k$-way parallelism for values up to $k = 6$. We can see that
/* Unroll loop by 2, 2-way parallelism */
void combine6(vec_ptr v, data_t *dest)
{
  long int i;
  long int length = vec_length(v);
  long int limit = length-1;
  data_t *data = get_vec_start(v);
  data_t acc0 = IDENT;
  data_t acc1 = IDENT;

  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i+1];
  }

  /* Finish any remaining elements */
  for (; i < length; i++) {
    acc0 = acc0 OP data[i];
  }
  *dest = acc0 OP acc1;
}

Figure 5.21  Unrolling loop by 2 and using two-way parallelism. This approach makes use of the pipelining capability of the functional units.

the CPEs for all of our combining cases improve with increasing values of \( k \). For integer multiplication, and for the floating-point operations, we see a CPE value of \( L/k \), where \( L \) is the latency of the operation, up to the throughput bound of 1.00. We also see integer addition reaching its throughput bound of 1.00 with \( k = 3 \). Of course, we also reached this bound for integer addition with standard unrolling.

Figure 5.22  CPE performance for \( k \)-way loop unrolling with \( k \)-way parallelism. All of the CPEs improve with this transformation, up to the limiting value of 1.00.
Figure 5.23  **Graphical representation of inner-loop code for combine6.** Each iteration has two mulss instructions, each of which is translated into a load and a mul operation.

Figure 5.24  **Abstracting combine6 operations as data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.23 to show the data dependencies between successive iterations (a). We see that there is no dependency between the two mul operations (b).

To understand the performance of combine6, we start with the code and operation sequence shown in Figure 5.23. We can derive a template showing the data dependencies between iterations through the process shown in Figure 5.24. As with combine5, the inner loop contains two mulss operations, but these instructions translate into mul operations that read and write separate registers, with no data dependency between them (Figure 5.24(b)). We then replicate this template \( n/2 \) times (Figure 5.25), modeling the execution of the function on a vector.
of length \( n \). We see that we now have two critical paths, one corresponding to computing the product of even-numbered elements (program value \( \text{acc0} \)) and one for the odd-numbered elements (program value \( \text{acc1} \)). Each of these critical paths contain only \( n/2 \) operations, thus leading to a CPE of 4.00/2. A similar analysis explains our observed CPE of \( L/2 \) for operations with latency \( L \) for the different combinations of data type and combining operation. Operationally, we are exploiting the pipelining capabilities of the functional unit to increase their utilization by a factor of 2. When we apply this transformation for larger values of \( k \), we find that we cannot reduce the CPE below 1.00. Once we reach this point, several of the functional units are operating at maximum capacity.

We have seen in Chapter 2 that two’s-complement arithmetic is commutative and associative, even when overflow occurs. Hence, for an integer data type, the result computed by \( \text{combine6} \) will be identical to that computed by \( \text{combine5} \).
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. Many compilers do loop unrolling automatically, but relatively few then introduce this form of parallelism.

On the other hand, floating-point multiplication and addition are not associative. Thus, combine5 and combine6 could produce different results due to rounding or overflow. Imagine, for example, a product computation in which all of the elements with even indices were numbers with very large absolute value, while those with odd indices were very close to 0.0. In such a case, product $PE_n$ might overflow, or $PO_n$ might underflow, even though computing product $P_n$ proceeds normally. In most real-life applications, however, such patterns are unlikely. Since most physical phenomena are continuous, numerical data tend to be reasonably smooth and well-behaved. Even when there are discontinuities, they do not generally cause periodic patterns that lead to a condition such as that sketched earlier. It is unlikely that multiplying the elements in strict order gives fundamentally better accuracy than does multiplying two groups independently and then multiplying those products together. For most applications, achieving a performance gain of $2 \times$ 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.

5.9.2 Reassociation Transformation

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

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

\[
\text{acc} = (\text{acc OP data[i]} \text{ OP data[i+1]});
\]

while in combine7 it is performed by the statement

\[
\text{acc} = \text{acc OP (data[i] OP data[i+1])};
\]

differing only in how two parentheses are placed. We call this a reassociation transformation, because the parentheses shift the order in which the vector elements are combined with the accumulated value acc.

To an untrained eye, the two statements may seem essentially the same, but when we measure the CPE, we get surprising results:
The integer multiplication case nearly matches the performance of the version with simple unrolling (combine5), while the floating-point cases match the performance of the version with parallel accumulators (combine6), doubling the performance relative to simple unrolling. (The CPE of 2.97 shown for double-precision multiplication is most likely the result of a measurement error, with the true value being 2.50. In our experiments, we found the measured CPEs for combine7 to be more variable than for the other functions.)

Figure 5.27 demonstrates the effect of applying the reassociation transformation to achieve $k$-way loop unrolling with reassociation. We can see that the CPEs for all of our combining cases improve with increasing values of $k$. For integer

```c
/* Change associativity of combining operation */
void combine7(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    long int limit = length - 1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OP (data[i] OP data[i+1]);
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Figure 5.26  Unrolling loop by 2 and then reassociating the combining operation. This approach also increases the number of operations that can be performed in parallel.
multiplication and for the floating-point operations, we see a CPE value of nearly $L/k$, where $L$ is the latency of the operation, up to the throughput bound of 1.00. We also see integer addition reaching CPE of 1.00 for $k = 3$, achieving both the throughput and the latency bounds.

Figure 5.28 illustrates how the code for the inner loop of combine7 (for the case of single-precision product) gets decoded into operations and the resulting data dependencies. We see that the load operations resulting from the `movss` and the first `mulss` instructions load vector elements $i$ and $i + 1$ from memory, and the first `mul` operation multiplies them together. The second `mul` operation then multiplies this result by the accumulated value `acc`. Figure 5.29 shows how we rearrange, refine, and abstract the operations of Figure 5.28 to get a template representing the data dependencies for one iteration (Figure 5.29(b)). As with the templates for combine5 and combine7, we have two load and two `mul` operations,
Section 5.9 Enhancing Parallelism

Figure 5.29 Abstracting combine operations as data-flow graph. We rearrange, simplify, and abstract the representation of Figure 5.28 to show the data dependencies between successive iterations (a). The first mul operation multiplies the two vector elements, while the second one multiplies the result by loop variable acc (b).

but only one of the mul operations forms a data-dependency chain between loop registers. When we then replicate this template \( n/2 \) times to show the computations performed in multiplying \( n \) vector elements (Figure 5.30), we see that we only have \( n/2 \) operations along the critical path. The first multiplication within each iteration can be performed without waiting for the accumulated value from the previous iteration. Thus, we reduce the minimum possible CPE by a factor of 2. As we increase \( k \), we continue to have only one operation per iteration along the critical path.

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

We can now explain the surprising improvement we saw with simple loop unrolling (combine5) for the case of integer multiplication. In compiling this code, gcc performed the reassociation that we have shown in combine7, and hence it achieved the same performance. It also performed the transformation for code with higher degrees of unrolling. gcc recognizes that it can safely perform this transformation for integer operations, but it also recognizes that it cannot transform the floating-point cases due to the lack of associativity. It would be gratifying to find that gcc performed this transformation recognizing that the resulting code would run faster, but unfortunately this seems not to be the case. In our experiments, we found that very minor changes to the C code caused gcc
Figure 5.30
Data-flow representation of combine7 operating on a vector of length $n$. We have a single critical path, but it contains only $n/2$ operations.

to associate the operations differently, sometimes causing the generated code to speed up, and sometimes to slow down, relative to what would be achieved by a straightforward compilation. Optimizing compilers must choose which factors they try to optimize, and it appears that gcc does not use maximizing instruction-level parallelism as one of its optimization criteria when selecting how to associate integer operations.

In summary, a reassociation transformation can reduce the number of operations along the critical path in a computation, resulting in better performance by better utilizing the pipelining capabilities of the functional units. Most compilers will not attempt any reassociations of floating-point operations, since these operations are not guaranteed to be associative. Current versions of gcc do perform reassociations of integer operations, but not always with good effects. In general, we have found that unrolling a loop and accumulating multiple values in parallel is a more reliable way to achieve improved program performance.
Practice Problem 5.8

Consider the following function for computing the product of an array of \( n \) integers. We have unrolled the loop by a factor of 3.

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

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

- \( r = ((r \times x) \times y) \times z \); /* A1 */
- \( r = (r \times (x \times y)) \times z \); /* A2 */
- \( r = r \times ((x \times y) \times z) \); /* A3 */
- \( r = r \times (x \times (y \times z)) \); /* A4 */
- \( r = (r \times x) \times (y \times z) \); /* A5 */

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

Web Aside OPT:SIMD  
Achieving greater parallelism with SIMD instructions

As described in Section 3.1, Intel introduced the SSE instructions in 1999, where SSE is the acronym for “Streaming SIMD Extensions,” and, in turn, SIMD (pronounced “sim-dee”) is the acronym for “Single-Instruction, Multiple-Data.” The idea behind the SIMD execution model is that each 16-byte XMM register can hold multiple values. In our examples, we consider the cases where they can hold either four integer or single-precision values, or two double-precision values. SSE instructions can then perform vector operations on these registers, such as adding or multiplying four or two sets of values in parallel. For example, if XMM register \( %xmm0 \) contains four single-precision floating-point numbers, which we denote \( a_0, \ldots, a_3 \), and \( %rcx \) contains the memory address of a sequence of four single-precision floating-point numbers, which we denote \( b_0, \ldots, b_3 \), then the instruction

\( \text{mulps \hspace{1em}} (%rcx), \hspace{1em} %xmm0 \)
will read the four values from memory and perform four multiplications in parallel, computing \( a_i \leftarrow a_i \cdot b_i \), for \( 0 \leq i \leq 3 \). We see that a single instruction is able to generate a computation over multiple data values, hence the term “SIMD.”

`gcc` supports extensions to the C language that let programmers express a program in terms of vector operations that can be compiled into the SIMD instructions of SSE. This coding style is preferable to writing code directly in assembly language, since `gcc` can also generate code for the SIMD instructions found on other processors.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>SSE + 8-way unrolling</td>
<td>0.25</td>
<td>0.55</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>0.25</td>
<td>0.50</td>
</tr>
</tbody>
</table>

As this chart shows, using SSE instructions lowers the throughput bound, and we have nearly achieved these bounds for all five cases. The throughput bound of 0.25 for integer addition and single-precision addition and multiplication is due to the fact that the SSE instruction can perform four of these in parallel, and it has an issue time of 1. The double-precision instructions can only perform two in parallel, giving a throughput bound of 0.50. The integer multiplication operation has a throughput bound of 0.50 for a different reason—although it can perform four in parallel, it has an issue time of 2. In fact, this instruction is only available for SSE versions 4 and higher (requiring command-line flag `"-msse4"`).

### 5.10 Summary of Results for Optimizing Combining Code

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>combine1</td>
<td>485</td>
<td>Abstract -O1</td>
<td>12.00</td>
<td>12.00</td>
</tr>
<tr>
<td>combine6</td>
<td>515</td>
<td>Unroll by ( \times 2 ), parallelism ( \times 2 )</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unroll by ( \times 5 ), parallelism ( \times 5 )</td>
<td>1.01</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Latency bound

|                         | 1.00 | 3.00 | 3.00 | 4.00 | 5.00 |

Throughput bound

|                         | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

By using multiple optimizations, we have been able to achieve a CPE close to 1.00 for all combinations of data type and operation using ordinary C code, a performance improvement of over 10X compared to the original version `combine1`. 

As covered in Web Aside opt:simd, we can improve performance even further by making use of gcc’s support for SIMD vector instructions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>SIMD code</td>
<td>SIMD + 8-way unrolling</td>
<td>0.25</td>
<td>0.55</td>
</tr>
<tr>
<td>Throughput bound</td>
<td></td>
<td>0.25</td>
<td>0.50</td>
</tr>
</tbody>
</table>

The processor can sustain up to four combining operations per cycle for integer and single-precision data, and two per cycle for double-precision data. This represents a performance of over 6 gigaflops (billions of floating-point operations per second) on a processor now commonly found in laptop and desktop machines.

Compare this performance to that of the Cray 1S, a breakthrough supercomputer introduced in 1976. This machine cost around $8 million and consumed 115 kilowatts of electricity to get its peak performance of 0.25 gigaflops, over 20 times slower than we measured here.

Several factors limit our performance for this computation to a CPE of 1.00 when using scalar instructions, and a CPE of either 0.25 (32-bit data) or 0.50 (64-bit data) when using SIMD instructions. First, the processor can only read 16 bytes from the data cache on each cycle, and then only by reading into an XMM register. Second, the multiplier and adder units can only start a new operation every clock cycle (in the case of SIMD instructions, each of these “operations” actually computes two or four sums or products). Thus, we have succeeded in producing the fastest possible versions of our combining function for this machine.

5.11 Some Limiting Factors

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

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

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

5.11.1 Register Spilling

The benefits of loop parallelism are limited by the ability to express the computation in assembly code. 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 can drop significantly. As an illustration, compare the performance of our parallel accumulator code for integer sum on x86-64 vs. IA32:

<table>
<thead>
<tr>
<th>Degree of unrolling</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>IA32</td>
<td>2.12</td>
</tr>
<tr>
<td>x86-64</td>
<td>2.00</td>
</tr>
</tbody>
</table>

We see that for IA32, the lowest CPE is achieved when just \( k = 4 \) values are accumulated in parallel, and it gets worse for higher values of \( k \). We also see that we cannot get down to the CPE of 1.00 achieved for x86-64.

Examining the IA32 code for the case of \( k = 5 \) shows the effect of the small number of registers with IA32:

```assembly
IA32 code. Unroll X5, accumulate X5, data_t = int, OP = +
i in %edx, data in %eax, limit at %ebp-20
.L291:
  loop:
  1   imull (%eax,%edx,4), %ecx   x0 = x0 * data[i]  
  2   movl -16(%ebp), %ebx       Get x1  
  3   imull 4(%eax,%edx,4), %ebx x1 = x1 * data[i+1]  
  4   movl %ebx, -16(%ebp)       Store x1  
  5   imull 8(%eax,%edx,4), %edi x2 = x2 * data[i+2]  
  6   imull 12(%eax,%edx,4), %esi x3 = x3 * data[i+3]  
  7   movl -28(%ebp), %ebx       Get x4  
  8   imull 16(%eax,%edx,4), %ebx x4 = x4 * daa[i+4]  
  9   movl %ebx, -28(%ebp)       Store x4  
  10  addl $5, %edx               i+= 5  
  11  cmpl %edx, -20(%ebp)       Compare limit:i  
  12  jg  .L291                 If >, goto loop
```

We see here that accumulator values acc1 and acc4 have been “spilled” onto the stack, at offsets \(-16\) and \(-28\) relative to \%ebp. In addition, the termination value limit is kept on the stack at offset \(-20\). The loads and stores associated with reading these values from memory and then storing them back negates any value obtained by accumulating multiple values in parallel.

We can now see the merit of adding eight additional registers in the extension of IA32 to x86-64. The x86-64 code is able to accumulate up to 12 values in parallel without spilling any registers.

### 5.11.2 Branch Prediction and Misprediction Penalties

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

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

In a processor that employs speculative execution, the processor begins executing the instructions at the predicted branch target. It does this in a way that avoids modifying any actual register or memory locations until the actual outcome has been determined. If the prediction is correct, the processor can then “commit” the results of the speculatively executed instructions by storing them in registers or memory. If the prediction is incorrect, the processor must discard all of the speculatively executed results and restart the instruction fetch process at the correct location. The misprediction penalty is incurred in doing this, because the instruction pipeline must be refilled before useful results are generated.

We saw in Section 3.6.6 that recent versions of x86 processors have conditional move instructions and that gcc can generate code that uses these instructions when compiling conditional statements and expressions, rather than the more traditional realizations based on conditional transfers of control. The basic idea for translating into conditional moves is to compute the values along both branches of a conditional expression or statement, and then use conditional moves to select the desired value. We saw in Section 4.5.10 that conditional move instructions can be implemented as part of the pipelined processing of ordinary instructions. There is no need to guess whether or not the condition will hold, and hence no penalty for guessing incorrectly.

How then can a C programmer make sure that branch misprediction penalties do not hamper a program’s efficiency? Given the 44 clock-cycle misprediction penalty we saw for the Intel Core i7, the stakes are very high. There is no simple answer to this question, but the following general principles apply.

Do Not Be Overly Concerned about Predictable Branches

We have seen that the effect of a mispredicted branch can be very high, but that does not mean that all program branches will slow a program down. In fact, the branch prediction logic found in modern processors is very good at discerning regular patterns and long-term trends for the different branch instructions. For example, the loop-closing branches in our combining routines would typically be predicted as being taken, and hence would only incur a misprediction penalty on the last time around.
As another example, consider the small performance gain we observed when shifting from combine2 to combine3, when we took the function get_vec_element out of the inner loop of the function, as is reproduced below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>combine2</td>
<td>486</td>
<td>Move vec_length</td>
<td>8.03</td>
<td>8.09</td>
</tr>
<tr>
<td>combine3</td>
<td>491</td>
<td>Direct data access</td>
<td>6.01</td>
<td>8.01</td>
</tr>
</tbody>
</table>

The CPE hardly changed, even though this function uses two conditionals to check whether the vector index is within bounds. These checks always determine that the index is within bounds, and hence they are highly predictable.

As a way to measure the performance impact of bounds checking, consider the following combining code, where we have modified the inner loop of combine4 by replacing the access to the data element with the result of performing an inline substitution of the code for get_vec_element. We will call this new version combine4b. This code performs bounds checking and also references the vector elements through the vector data structure.

```c
/* Include bounds check in loop */
void combine4b(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t acc = IDENT;

    for (i = 0; i < length; i++) {
        if (i >= 0 && i < v->len) {
            acc = acc OP v->data[i];
        }
    }

    *dest = acc;
}
```

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Method</th>
<th>Integer</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>combine4</td>
<td>493</td>
<td>No bounds checking</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>combine4b</td>
<td>493</td>
<td>Bounds checking</td>
<td>4.00</td>
<td>4.00</td>
</tr>
</tbody>
</table>

Although the performance of the version with bounds checking is not quite as good, it increases the CPE by at most 2 clock cycles. This is a fairly small difference, considering that the bounds checking code performs two conditional branches.
and it also requires a load operation to implement the expression \( v \rightarrow \text{len} \). The processor is able to predict the outcomes of these branches, and so none of this evaluation has much effect on the fetching and processing of the instructions that form the critical path in the program execution.

**Write Code Suitable for Implementation with Conditional Moves**

Branch prediction is only reliable for regular patterns. Many tests in a program are completely unpredictable, dependent on arbitrary features of the data, such as whether a number is negative or positive. For these, the branch prediction logic will do very poorly, possibly giving a prediction rate of 50%—no better than random guessing. (In principle, branch predictors can have prediction rates less than 50%, but such cases are very rare.) For inherently unpredictable cases, program performance can be greatly enhanced if the compiler is able to generate code using conditional data transfers rather than conditional control transfers. This cannot be controlled directly by the C programmer, but some ways of expressing conditional behavior can be more directly translated into conditional moves than others.

We have found that `gcc` is able to generate conditional moves for code written in a more “functional” style, where we use conditional operations to compute values and then update the program state with these values, as opposed to a more “imperative” style, where we use conditionals to selectively update program state.

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

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

```c
/* Rearrange two vectors so that for each i, b[i] >= a[i] */
void minmax1(int a[], int b[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] > b[i]) {
            int t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

Our measurements for this function on random data show a CPE of around 14.50 for random data, and 3.00–4.00 for predictable data, a clear sign of a high misprediction penalty.
A functional style of implementing this function is to compute the minimum and maximum values at each position $i$ and then assign these values to $a[i]$ and $b[i]$, respectively:

```c
/* Rearrange two vectors so that for each i, b[i] >= a[i] */
void minmax2(int a[], int b[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        int min = a[i] < b[i] ? a[i] : b[i];
        int max = a[i] < b[i] ? b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

Our measurements for this function show a CPE of around 5.0 regardless of whether the data are arbitrary or predictable. (We also examined the generated assembly code to make sure that it indeed used conditional moves.)

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

**Practice Problem 5.9**

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

```c
void merge(int src1[], int src2[], int dest[], int n) {
    int i1 = 0;
    int i2 = 0;
    int id = 0;
    while (i1 < n && i2 < n) {
        if (src1[i1] < src2[i2])
            dest[id++] = src1[i1++];
        else
            dest[id++] = src2[i2++];
    }
    while (i1 < n)
        dest[id++] = src1[i1++];
    while (i2 < n)
        dest[id++] = src2[i2++];
}
```
The branches caused by comparing variables \( i_1 \) and \( i_2 \) to \( n \) have good prediction performance—the only mispredictions occur when they first become false. The comparison between values \( \text{src1}[i_1] \) and \( \text{src2}[i_2] \) (line 6), on the other hand, is highly unpredictable for typical data. This comparison controls a conditional branch, yielding a CPE (where the number of elements is \( 2n \)) of around 17.50.

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

5.12 Understanding Memory Performance

All of the code we have written thus far, and all the tests we have run, access relatively small amounts of memory. For example, the combining routines were measured over vectors of length less than 1000 elements, requiring no more than 8000 bytes of data. All modern processors contain one or more cache memories to provide fast access to such small amounts of memory. In this section, we will further investigate the performance of programs that involve load (reading from memory into registers) and store (writing from registers to memory) operations, considering only the cases where all data are held in cache. In Chapter 6, we go into much more detail about how caches work, their performance characteristics, and how to write code that makes best use of caches.

As Figure 5.11 shows, modern processors have dedicated functional units to perform load and store operations, and these units have internal buffers to hold sets of outstanding requests for memory operations. For example, the Intel Core i7 load unit’s buffer can hold up to 48 read requests, while the store unit’s buffer can hold up to 32 write requests [99]. Each of these units can typically initiate one operation every clock cycle.

5.12.1 Load Performance

The performance of a program containing load operations depends on both the pipelining capability and the latency of the load unit. In our experiments with combining operations on a Core i7, we saw that the CPE never got below 1.00, except when using SIMD operations. One factor limiting the CPE for our examples is that they all require reading one value from memory for each element computed. Since the load unit can only initiate one load operation every clock cycle, the CPE cannot be less than 1.00. For applications where we must load \( k \) values for every element computed, we can never achieve a CPE lower than \( k \) (see, for example, Problem 5.17).

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

To determine the latency of the load operation on a machine, we can set up a computation with a sequence of load operations, where the outcome of one
Chapter 5  Optimizing Program Performance

determines the address for the next. As an example, consider the function `list_len` in Figure 5.31, which computes the length of a linked list. In the loop of this function, each successive value of variable `ls` depends on the value read by the pointer reference `ls->next`. Our measurements show that function `list_len` has a CPE of 4.00, which we claim is a direct indication of the latency of the load operation. To see this, consider the assembly code for the loop. (We show the x86-64 version of the code. The IA32 code is very similar.)

```
    len in %eax, ls in %rdi
    .L11:  loop:
        addl $1, %eax     Increment len
        movq (%rdi), %rdi  ls = ls->next
        testq %rdi, %rdi   Test ls
        jne .L11           If nonnull, goto loop
```

The `movq` instruction on line 3 forms the critical bottleneck in this loop. Each successive value of register `%rdi` depends on the result of a load operation having the value in `%rdi` as its address. Thus, the load operation for one iteration cannot begin until the one for the previous iteration has completed. The CPE of 4.00 for this function is determined by the latency of the load operation.

5.12.2  Store Performance

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

As with the load operation, in most cases, the store operation can operate in a fully pipelined mode, beginning a new store on every cycle. For example, consider the 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. By unrolling the loop four times, as shown in the code for clear_array_4, we achieve a CPE of 1.00. Thus, we have achieved the optimum 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 cannot create a data dependency. Only a load operation is affected by the result of a store operation, since only a load can read back the memory value that has been written by the store. The function write_read shown in Figure 5.33 illustrates the potential interactions between loads and stores. This figure also shows two example executions of this function, when it is called for a two-element array \( a \), with initial contents \(-10\) and \(17\), and with argument \( \text{cnt} \) equal to 3. These executions illustrate some subtleties of the load and store operations.

In Example A of Figure 5.33, argument \( \text{src} \) is a pointer to array element \( a[0] \), while \( \text{dest} \) is a pointer to array element \( a[1] \). In this case, each load by the pointer reference \( *\text{src} \) 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 \( \text{src} \) is not affected by the write to \( \text{dest} \). Measuring this example over a larger number of iterations gives a CPE of 2.00.

In Example B of Figure 5.33, both arguments \( \text{src} \) and \( \text{dest} \) are pointers to array element \( a[0] \). In this case, each load by the pointer reference \( *\text{src} \) will yield the value stored by the previous execution of the pointer reference \( *\text{dest} \). As a consequence, a series of ascending values will be stored in this location. In general,
Chapter 5  Optimizing Program Performance

```c
/* Write to dest, read from src */
void write_read(int *src, int *dest, int n)
{
int cnt = n;
int val = 0;
while (cnt--)
{
    *dest = val;
    val = (*src)+1;
}
}
```

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

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Iter. 1</th>
<th>Iter. 2</th>
<th>Iter. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>-10 17</td>
<td>-10 0</td>
<td>-10 -9</td>
<td>-10 -9</td>
</tr>
<tr>
<td>val</td>
<td>0</td>
<td>-9</td>
<td>-9</td>
<td>-9</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Iter. 1</th>
<th>Iter. 2</th>
<th>Iter. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>-10 17</td>
<td>0 17</td>
<td>1 17</td>
<td>2 17</td>
</tr>
<tr>
<td>val</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

*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.

If function `write_read` is called with arguments `src` and `dest` pointing to the same memory location, and with argument `cnt` having some value `n > 0`, the net effect is to set the location to `n - 1`. This example illustrates a phenomenon we will call a write/read dependency—the outcome of a memory read depends on a recent memory write. Our performance measurements show that Example B has a CPE of 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 the other, we must take a more detailed look at the load and store execution units, as shown in Figure 5.34. The store unit 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
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.

A load operation occurs, it must check the entries in the store buffer for matching addresses. If it finds a match (meaning that any of the bytes being written have the same address as any of the bytes being read), it retrieves the corresponding data entry as the result of the load operation.

Figure 5.35 shows the assembly code for the inner loop of `write_read`, and a graphical representation of the operations generated by the instruction decoder. The instruction `movl %eax, (%ecx)` is translated into two operations: The `s_addr` instruction computes the address for the store operation, creates an entry in the store buffer, and sets the address field for that entry. The `s_data` operation sets the data field for the entry. As we will see, the fact that these two computations are performed independently can be important to program performance.

In addition to the data dependencies between the operations caused by the writing and reading of registers, the arcs on the right of the operators denote a set of implicit dependencies for these operations. In particular, the address computation of the `s_addr` operation must clearly precede the `s_data` operation. In addition, the load operation generated by decoding the instruction `movl (%ebx),`
Abstracting the operations for write_read. We first rearrange the operations of Figure 5.35 (a) and then show only those operations that use values from one iteration to produce new values for the next (b).

%eax must check the addresses of any pending store operations, creating a data dependency between it and the $s\_addr$ operation. The figure shows a dashed arc between the $s\_data$ and load operations. This dependency is conditional: if the two addresses match, the load operation must wait until the $s\_data$ has deposited its result into the store buffer, but if the two addresses differ, the two operations can proceed independently.

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

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

We can now understand the performance characteristics of function write_read. Figure 5.37 illustrates the data dependencies formed by multiple iterations of its inner loop. For the case of Example A of Figure 5.33, with differing source and destination addresses, the load and store operations can proceed independently, and hence the only critical path is formed by the decrementing of variable $cnt$. This would lead us to predict a CPE of just 1.00, rather than the measured CPE of 2.00. We have found similar behavior for any function where data are both being stored and loaded within a loop. Apparently the effort to compare load addresses with those of the pending store operations forms an additional bottleneck. For
the case of Example B, with matching source and destination addresses, the data dependency between the \texttt{s\_data} and \texttt{load} instructions causes a critical path to form involving data being stored, loaded, and incremented. We found that these three operations in sequence require a total of 6 clock cycles.

As these two examples show, the implementation of memory operations involves many subtleties. With operations on registers, the processor can determine which instructions will affect which others as they are being decoded into operations. With memory operations, on the other hand, the processor cannot predict which will affect which others until the load and store addresses have been computed. Efficient handling of memory operations is critical to the performance of many programs. The memory subsystem makes use of many optimizations, such as the potential parallelism when operations can proceed independently.
Practice Problem 5.10

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

```c
void copy_array(int *src, int *dest, int n)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = src[i];
}
```

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 2.00, 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)`?

Practice Problem 5.11

We saw that our measurements of the prefix-sum function `psum1` (Figure 5.1) yield a CPE of 10.00 on a machine where the basic operation to be performed, floating-point addition, has a latency of just 3 clock cycles. Let us try to understand why our function performs so poorly.

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

```
.psum1. a in %rdi, p in %rsi, i in %rax, cnt in %rdx
.L5:          loop:
    movss -4(%rsi,%rax,4), %xmm0         Get p[i-1]
    addss (%rdi,%rax,4), %xmm0          Add a[i]
    movss %xmm0, (%rsi,%rax,4)          Store at p[i]
    addq $1, %rax                        Increment i
    cmpq %rax, %rdx                      Compare cnt:i
    jg .L5                               If >, goto loop
```

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

Explain why the CPE is so high. (You may not be able to justify the exact CPE, but you should be able to describe why it runs more slowly than one might expect.)
**Practice Problem 5.12**

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

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

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

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.
   - Eliminate excessive function calls. Move computations out of loops when possible. Consider selective compromises of program modularity to gain greater efficiency.
   - Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.

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

A final word of advice to the reader is to be vigilant to avoid introducing errors as you rewrite programs in the interest of efficiency. It is very easy to make mistakes when introducing new variables, changing loop bounds, and making the code more complex overall. One useful technique is to use checking code to test each version of a function as it is being optimized, to ensure no bugs are introduced during this process. Checking code applies a series of tests to the new versions of a function and makes sure they yield the same results as the original. The set of test cases must become more extensive with highly optimized code, since there are more cases to consider. For example, checking code that uses loop unrolling requires testing for many different loop bounds to make sure it handles all of the different possible numbers of single-step iterations required at the end.
Chapter 5 Optimizing Program Performance

5.14 Identifying and Eliminating Performance Bottlenecks

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

5.14.1 Program Profiling

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

Unix systems provide the profiling program gprof. This program generates two forms of information. First, it determines how much CPU time was spent for each of the functions in the program. Second, it computes a count of how many times each function gets called, categorized by which function performs the call. Both forms of information can be quite useful. The timings give a sense of the relative importance of the different functions in determining the overall run time. The calling information allows us to understand the dynamic behavior of the program.

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

1. The program must be compiled and linked for profiling. With gcc (and other C compilers) this involves simply including the run-time flag ‘-pg’ on the command line:

   unix> gcc -O1 -pg prog.c -o prog

2. The program is then executed as usual:

   unix> ./prog file.txt

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

3. gprof is invoked to analyze the data in gmon.out.

   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 listing shows this part of the report for the three most time-consuming functions in a program:
### Identifying and Eliminating Performance Bottlenecks

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>seconds</th>
<th>calls</th>
<th>s/call</th>
<th>total call time</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.58</td>
<td>173.05</td>
<td>173.05</td>
<td>1</td>
<td>173.05</td>
<td>173.05</td>
<td></td>
<td>sort_words</td>
</tr>
<tr>
<td>2.36</td>
<td>177.24</td>
<td>4.19</td>
<td>965027</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>find_ele_rec</td>
</tr>
<tr>
<td>0.12</td>
<td>177.46</td>
<td>0.22</td>
<td>12511031</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Strlen</td>
</tr>
</tbody>
</table>

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 173.05 seconds, while the function `find_ele_rec` was called 965,027 times (not including recursive calls), requiring a total of 4.19 seconds. Function `Strlen` computes the length of a string by calling the library function `strlen`. Library function calls are normally not shown in the results by `gprof`. Their times are usually reported as part of the function calling them. By creating the “wrapper function” `Strlen`, we can reliably track the calls to `strlen`, showing that it was called 12,511,031 times, but only requiring a total of 0.22 seconds.

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

```
158655725       find_ele_rec [5]
               insert_string [4]
[5]    2.4  4.19  0.02  965027/965027
0.01  0.01  363039/363039     new_ele [10]
0.00  0.01  363039/363039     save_string [13]
158655725       find_ele_rec [5]
```

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

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

Some properties of `gprof` are worth noting:

- The timing is not very precise. It is based on a simple *interval counting* scheme in which the compiled program maintains a counter for each function recording the time spent executing that function. The operating system causes the program to be interrupted at some regular time interval $\delta$. Typical values of
5.14.2 Using a Profile to Guide Optimization

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

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

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

1. Each word is read from the file and converted to lowercase. Our initial version used the function `lower1` (Figure 5.7), which we know to have quadratic run time due to repeated calls to `strlen`.

2. A hash function is applied to the string to create a number between 0 and s − 1, for a hash table with s buckets. Our initial function simply summed the ASCII codes for the characters modulo s.

3. Each hash bucket is organized as a linked list. The program scans down this list looking for a matching entry. If one is found, the frequency for this n-gram
Section 5.14 Identifying and Eliminating Performance Bottlenecks

is incremented. Otherwise, a new list element is created. Our initial version performed this operation recursively, inserting new elements at the end of the list.

4. Once the table has been generated, we sort all of the elements according to the frequencies. Our initial version used insertion sort.

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

**Sort:** Sorting n-grams by frequency

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

**Lower:** Converting strings to lowercase
Chapter 5 Optimizing Program Performance

**Strlen:** Computing string lengths

**Hash:** Computing the hash function

**Rest:** The sum of all other functions

As part (a) of the figure shows, our initial version required nearly 3 minutes, with most of the time spent sorting. This is not surprising, since insertion sort has quadratic run time, and the program sorted 363,039 values.

In our next version, we performed sorting using the library function `qsort`, which is based on the quicksort algorithm, having run time $O(n \log n)$. This version is labeled “Quicksort” in the figure. The more efficient sorting algorithm reduces the time spent sorting to become negligible, and the overall run time to around 4.7 seconds. Part (b) of the figure shows the times for the remaining version on a scale where we can see them more clearly.

With improved sorting, we now find that list scanning becomes the bottleneck. Thinking that the inefficiency is due to the recursive structure of the function, we replaced it by an iterative one, shown as “Iter first.” Surprisingly, the run time increases to around 5.9 seconds. On closer study, we find a subtle difference between the two list functions. The recursive version inserted new elements at the end of the list, while the iterative one inserted them at the front. To maximize performance, we want the most frequent n-grams to occur near the beginnings of the lists. That way, the function will quickly locate the common cases. Assuming that n-grams are spread uniformly throughout the document, we would expect the first occurrence of a frequent one to come before that of a less frequent one. By inserting new n-grams at the end, the first function tended to order n-grams in descending order of frequency, while the second function tended to do just the opposite. We therefore created a third list-scanning function that uses iteration, but inserts new elements at the end of this list. With this version, shown as “Iter last,” the time dropped to around 4.2 seconds, slightly better than with the recursive version. These measurements demonstrate the importance of running experiments on a program as part of an optimization effort. We initially assumed that converting recursive code to iterative code would improve its performance and did not consider the distinction between adding to the end or to the beginning of a list.

Next, we consider the hash table structure. The initial version had only 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 363,039 entries, this would imply an average load of $\frac{363039}{1021} = 355.6$. That explains why so much of the time is spent performing list operations—the searches involve testing a significant number of candidate n-grams. It also explains why the performance is so sensitive to the list ordering. We then increased the number of buckets to 199,999, reducing the average load to 1.8. Oddly enough, however, our overall run time only drops to 3.9 seconds, a difference of only 0.3 seconds.

On further inspection, we can see that the minimal performance gain with a larger table was due to a poor choice of hash function. Simply summing the character codes for a string does not produce a very wide range of values. In particular,
Section 5.14 Identifying and Eliminating Performance Bottlenecks

the maximum code value for a letter is 122, and so a string of $n$ characters will generate a sum of at most $122n$. The longest bigram in our document, “honorificabilitudinitatibus thou,” sums to just 3371, and so most of the buckets in our hash table will go unused. In addition, a commutative hash function, such as addition, does not differentiate among the different possible orderings of characters with a string. For example, the words “rat” and “tar” will generate the same sums.

We switched to a hash function that uses shift and Exclusive-Or operations. With this version, shown as “Better hash,” the time drops to 0.4 seconds. A more systematic approach would be to study the distribution of keys among the buckets more carefully, making sure that it comes close to what one would expect if the hash function had a uniform output distribution.

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

With this exercise, we have shown that code profiling can help drop the time required for a simple application from nearly 3 minutes down to well under 1 second. The profiler helps us focus our attention on the most time-consuming parts of the program and also provides useful information about the procedure call structure. Some of the bottlenecks in our code, such as using a quadratic sort routine, are easy to anticipate, while others, such as whether to append to the beginning or end of a list, emerge only through a careful analysis.

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

5.14.3 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 has come to be known as 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 in which executing some application requires time
Suppose some part of the system requires a fraction \( \alpha \) of this time, and that we improve its performance by a factor of \( k \). That is, the component originally required time \( \alpha T_{\text{old}} \), and it now requires time \((\alpha T_{\text{old}})/k\). The overall execution time would thus be

\[
T_{\text{new}} = (1 - \alpha)T_{\text{old}} + \frac{\alpha T_{\text{old}}}{k}
\]

\[
= T_{\text{old}}[(1 - \alpha) + \alpha/k]
\]

From this, we can compute the speedup \( S = T_{\text{old}}/T_{\text{new}} \) as

\[
S = \frac{1}{(1 - \alpha) + \alpha/k}
\]

(5.4)

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 \((k = 3)\). 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 5.13**

Suppose you work as a truck driver, and you have been hired to carry a load of potatoes from Boise, Idaho, to Minneapolis, Minnesota, a total distance of 2500 kilometers. You estimate you can average 100 km/hr driving within the speed limits, requiring a total of 25 hours for the trip.

A. You hear on the news that Montana has just abolished its speed limit, which constitutes 1500 km of the trip. Your truck can travel at 150 km/hr. What will be your speedup for the trip?

B. You can buy a new turbocharger for your truck at [www.fasttrucks.com](http://www.fasttrucks.com). They stock a variety of models, but the faster you want to go, the more it will cost. How fast must you travel through Montana to get an overall speedup for your trip of 5/3?

**Practice Problem 5.14**

The marketing department at your company has promised your customers that the next software release will show a 2\(\times\) 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 effect of setting \( k \) to \( \infty \). That is, we are able to take some part of the system and speed it up to the point at which it takes a negligible amount of time. We then get

\[
S_\infty = \frac{1}{(1 - \alpha)}
\]  

(5.5)

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 173.05 of its 177.57 seconds performing insertion sort, giving \( \alpha = 0.975 \). With quicksort, the time spent sorting becomes negligible, giving a predicted speedup of 39.3. In fact, the actual measured speedup was a bit less: 173.05/4.72 = 37.6, due to inaccuracies in the profiling measurements. 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 a company trying to reduce the cost of manufacturing razor blades, or a student trying to improve his or her grade point average. Perhaps it is most meaningful in the world of computers, where we routinely improve performance by factors of 2 or more. Such high factors can only be achieved by optimizing large parts of a system.

### 5.15 Summary

Although most presentations on code optimization describe how compilers can generate efficient code, much can be done by an application programmer to assist the compiler in this task. No compiler can replace an inefficient algorithm or data structure by a good one, and so these aspects of program design should remain a primary concern for programmers. We also have seen that optimization blockers, such as memory aliasing and procedure calls, seriously restrict the ability of compilers to perform extensive optimizations. Again, the programmer must take primary responsibility for eliminating these. These should simply be considered parts of good programming practice, since they serve to eliminate unneeded work.

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

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

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

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

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**

Our focus has been to describe code optimization from the programmer’s perspective, demonstrating how to write code that will make it easier for compilers to generate efficient code. An extended paper by Chellappa, Franchetti, and Püschel [19] takes a similar approach, but goes into more detail with respect to the processor’s characteristics.

Many publications describe code optimization from a compiler’s perspective, formulating ways that compilers can generate more efficient code. Muchnick’s book is considered the most comprehensive [76]. Wadleigh and Crawford’s book on software optimization [114] covers some of the material we have presented, but it also describes the process of getting high performance on parallel machines. An early paper by Mahlke et al. [71] describes how several techniques developed for compilers that map programs onto parallel machines can be adapted to exploit the instruction-level parallelism of modern processors. This paper covers the code transformations we presented, including loop unrolling, multiple accumulators (which they refer to as accumulator variable expansion), and reassociation (which they refer to as tree height reduction).

Our presentation of the operation of an out-of-order processor is fairly brief and abstract. More complete descriptions of the general principles can be found in
advanced computer architecture textbooks, such as the one by Hennessy and Pat-
terson [49, Ch. 2–3]. Shen and Lipasti’s book [96] provides an in-depth treatment
of modern processor design.

Amdahl’s law is presented in most books on computer architecture. With its
major focus on quantitative system evaluation, Hennessy and Patterson’s book
[49, Ch. 1] provides a particularly good treatment of the subject.

Homework Problems

5.15 ◆◆
Suppose we wish to write a procedure that computes the inner product of two
vectors \( u \) and \( v \). An abstract version of the function has a CPE of 16–17 with x86-
64 and 26–29 with IA32 for integer, single-precision, and double-precision 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:

```c
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long int i;
    int length = vec_length(u);
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;
    for (i = 0; i < length; i++) {
        sum = sum + udata[i] * vdata[i];
    }
    *dest = sum;
}
```

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

```
inner4: data_t = float  
udata in %rbx, vdata in %rax, limit in %rcx,  
i in %rdx, sum in %xmm1
.L87:
.loop:
1    movss (%rbx,%rdx,4), %xmm0   Get udata[i]
2    mulss (%rax,%rdx,4), %xmm0   Multiply by vdata[i]
3    addss %xmm0, %xmm1   Add to sum
4    addq $1, %rdx   Increment i
5    cmpq %rcx, %rdx   Compare i:limit
6    jl .L87   If <, goto loop
```

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

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

B. For data type \texttt{float}, what lower bound on the CPE is determined by the critical path?

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

D. Explain how the two floating-point versions can have CPEs of 3.00, even though the multiplication operation requires either 4 or 5 clock cycles.

5.16

Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling.

For x86-64, our measurements of the unrolled version give a CPE of 2.00 for integer data but still 3.00 for both single and double precision.

A. Explain why any version of any inner product procedure cannot achieve a CPE less than 2.00.

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

5.17

Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling with four parallel accumulators. Our measurements for this function with x86-64 give a CPE of 2.00 for all types of data.

A. What factor limits the performance to a CPE of 2.00?

B. Explain why the version with integer data on IA32 achieves a CPE of 2.75, worse than the CPE of 2.25 achieved with just four-way loop unrolling.

5.18

Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling along with reassociation to enable greater parallelism. Our measurements for this function give a CPE of 2.00 with x86-64 and 2.25 with IA32 for all types of data.

5.19

The library function \texttt{memset} has the following prototype:

\begin{verbatim}
void *memset(void *s, int c, size_t n);
\end{verbatim}

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

The following is a straightforward implementation of memset:

```c
/* Basic implementation of memset */
void *basic_memset(void *s, int c, size_t n)
{
    size_t cnt = 0;
    unsigned char *schar = s;
    while (cnt < n) {
        *schar++ = (unsigned char) c;
        cnt++;
    }
    return s;
}
```

Implement a more efficient version of the function by using a word of data type unsigned long to pack four (for IA32) or eight (for x86-64) copies of c, and then step through the region using word-level writes. You might find it helpful to do additional loop unrolling as well. On an Intel Core i7 machine, we were able to reduce the CPE from 2.00 for the straightforward implementation to 0.25 for IA32 and 0.125 for x86-64, i.e., writing either 4 or 8 bytes on every clock cycle.

Here are some additional guidelines. In this discussion, let \( K \) denote the value of \( \text{sizeof(unsigned long)} \) for the machine on which you run your program.

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

5.20 ◆◆◆
We considered the task of polynomial evaluation in Problems 5.5 and 5.6, with both a direct evaluation and an evaluation by Horner’s method. Try to write faster versions of the function using the optimization techniques we have explored, including loop unrolling, parallel accumulation, and reassociation. You will find many different ways of mixing together Horner’s scheme and direct evaluation with these optimization techniques.
Ideally, you should be able to reach a CPE close to the number of cycles between successive floating-point additions and multiplications with your machine (typically 1). At the very least, you should be able to achieve a CPE less than the latency of floating-point addition for your machine.

5.21 ◆◆◆
In Problem 5.12, we were able to reduce the CPE for the prefix-sum computation to 3.00, limited by the latency of floating-point addition on this machine. Simple loop unrolling does not improve things.

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

5.22 ◆
Suppose you are given the task of improving the performance of a program consisting of three parts. Part A requires 20% of the overall run time, part B requires 30%, and part C requires 50%. You determine that for $1000 you could either speed up part B by a factor of 3.0 or part C by a factor of 1.5. Which choice would maximize performance?

Solutions to Practice Problems

Solution to Problem 5.1 (page 478)
This problem illustrates some of the subtle effects of memory aliasing.

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

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

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

Solution to Problem 5.2 (page 482)
This problem illustrates the relationship between CPE and absolute performance. It can be solved using elementary algebra. We find that for \( n \leq 2 \), Version 1 is the fastest. Version 2 is fastest for \( 3 \leq n \leq 7 \), and Version 3 is fastest for \( n \geq 8 \).

Solution to Problem 5.3 (page 490)
This is a simple exercise, but it is important to recognize that the four statements of a `for` loop—initial, test, update, and body—get executed different numbers of times.
Solutions to Practice Problems

<table>
<thead>
<tr>
<th>Code</th>
<th>min</th>
<th>max</th>
<th>incr</th>
<th>square</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>1</td>
<td>91</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>B.</td>
<td>91</td>
<td>1</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>C.</td>
<td>1</td>
<td>1</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

**Solution to Problem 5.4 (page 494)**

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

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

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

```c
/* Make sure dest updated on each iteration */
void combine3w(vec_ptr v, data_t *dest) {
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
        *dest = acc;
    }
}
```

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

C. This transformation can be made without changing the program behavior, because, with the exception of the first iteration, the value read from dest at the beginning of each iteration will be the same value written to this register at the end of the previous iteration. Therefore, the combining instruction can simply use the value already in %xmm0 at the beginning of the loop.

**Solution to Problem 5.5 (page 507)**

Polynomial evaluation is a core technique for solving many problems. For example, polynomial functions are commonly used to approximate trigonometric functions in math libraries.
A. The function performs $2n$ multiplications and $n$ additions.

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

Solution to Problem 5.6 (page 508)
This problem demonstrates that minimizing the number of operations in a computation may not improve its performance.

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

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

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

Solution to Problem 5.7 (page 510)
The following code directly follows the rules we have stated for unrolling a loop by some factor $k$:

```c
1   void unroll5(vec_ptr v, data_t *dest)
2   {
3       long int i;
4       long int length = vec_length(v);
5       long int limit = length-4;
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       /* Combine 5 elements at a time */
10      for (i = 0; i < limit; i+=5) {
11          acc = acc OP data[i] OP data[i+1];
12          acc = acc OP data[i+2] OP data[i+3];
13          acc = acc OP data[i+4];
14      }
15
16      /* Finish any remaining elements */
17      for (; i < length; i++) {
18          acc = acc OP data[i];
```

Solution to Problem 5.8 (page 523)
This problem demonstrates how small changes in a program can yield dramatic performance differences, especially on a machine with out-of-order execution. Figure 5.39 diagrams the three multiplication operations for a single iteration of the function. In this figure, the operations shown as blue boxes are along the critical path—they need to be computed in sequence to compute a new value for loop variable r. The operations shown as light boxes can be computed in parallel with the critical path operations. For a loop with c operations along the critical path, each iteration will require a minimum of 5c clock cycles and will compute the product for three elements, giving a lower bound on the CPE of 5c/3. This implies lower bounds of 5.00 for A1, 3.33 for A2 and A5, and 1.67 for A3 and A4.

We ran these functions on an Intel Core i7, and indeed obtained CPEs of 5.00 for A1, and 1.67 for A3 and A4. For some reason, A2 and A5 achieved CPEs of just 3.67, indicating that the functions required 11 clock cycles per iteration rather than the predicted 10.

Solution to Problem 5.9 (page 530)
This is another demonstration that a slight change in coding style can make it much easier for the compiler to detect opportunities to use conditional moves:

```c
while (i1 < n && i2 < n) {
    int v1 = src1[i1];
    int v2 = src2[i2];
    int take1 = v1 < v2;
    dest[id++] = take1 ? v1 : v2;
    i1 += take1;
    i2 += (1-take1);
}
```
We measured a CPE of around 11.50 for this version of the code, a significant improvement over the original CPE of 17.50.

**Solution to Problem 5.10 (page 538)**

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

A. It will set each element $a[i]$ to $i + 1$, for $0 \leq i \leq 998$.
B. It will set each element $a[i]$ to 0, for $1 \leq i \leq 999$.
C. In the second case, the load of one iteration depends on the result of the store from the previous iteration. Thus, there is a write/read dependency between successive iterations. It is interesting to note that the CPE of 5.00 is 1 less than we measured for Example B of function `write_read`. This is due to the fact that `write_read` increments the value before storing it, requiring one clock cycle.
D. It will give a CPE of 2.00, the same as for Example A, since there are no dependencies between stores and subsequent loads.

**Solution to Problem 5.11 (page 538)**

We can see that this function has a write/read dependency between successive iterations—the destination value $p[i]$ on one iteration matches the source value $p[i-1]$ on the next.

**Solution to Problem 5.12 (page 539)**

Here is a revised version of the function:

```c
void psum1a(float a[], float p[], long int n) {
    long int i;
    /* last_val holds p[i-1]; val holds p[i] */
    float last_val, val;
    last_val = p[0] = a[0];
    for (i = 1; i < n; i++) {
        val = last_val + a[i];
        p[i] = val;
        last_val = val;
    }
}
```

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

This version compiles to the following assembly code:

```
 psum1a. a in %rdi, p in %rsi, i in %rax, cnt in %rdx, last_val in %xmm0
   .L18: loop:
   addss (%rdi,%rax,4), %xmm0 last_val = val = last_val + a[i]
```
This code holds last_val in %xmm0, avoiding the need to read p[i-1] from memory, and thus eliminating the write/read dependency seen in psum1.

Solution to Problem 5.13 (page 546)
This problem illustrates that Amdahl’s law applies to more than just computer systems.

A. In terms of Equation 5.4, we have $\alpha = 0.6$ and $k = 1.5$. More directly, traveling the 1500 kilometers through Montana will require 10 hours, and the rest of the trip also requires 10 hours. This will give a speedup of $25/(10 + 10) = 1.25$.

B. In terms of Equation 5.4, we have $\alpha = 0.6$, and we require $S = 5/3$, from which we can solve for $k$. More directly, to speed up the trip by $5/3$, we must decrease the overall time to 15 hours. The parts outside of Montana will still require 10 hours, so we must drive through Montana in 5 hours. This requires traveling at 300 km/hr, which is pretty fast for a truck!

Solution to Problem 5.14 (page 546)
Amdahl’s law is best understood by working through some examples. This one requires you to look at Equation 5.4 from an unusual perspective.

This problem is a simple application of the equation. You are given $S = 2$ and $\alpha = 0.8$, and you must then solve for $k$:

$$2 = \frac{1}{(1 - 0.8) + 0.8/k}$$

$$0.4 + 1.6/k = 1.0$$

$$k = 2.67$$
This page intentionally left blank
CHAPTER 6

The Memory Hierarchy

6.1 Storage Technologies  561
6.2 Locality  586
6.3 The Memory Hierarchy  591
6.4 Cache Memories  596
6.5 Writing Cache-friendly Code  615
6.6 Putting It Together: The Impact of Caches on Program Performance  620
6.7 Summary  629

Bibliographic Notes  630
Homework Problems  631
Solutions to Practice Problems  642
To this point in our study of systems, we have relied on a simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU. In our simple model, the memory system is a linear array of bytes, and the CPU can access each memory location in a constant amount of time. While this is an effective model as far as it goes, it does not reflect the way that modern systems really work.

In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. CPU registers hold the most frequently used data. Small, fast cache memories nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because well-written programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data your program needs are stored in a CPU register, then they can be accessed in zero cycles during the execution of the instruction. If stored in a cache, 1 to 30 cycles. If stored in main memory, 50 to 200 cycles. And if stored in disk tens of millions of cycles!

Here, then, is a fundamental and enduring idea in computer systems: if you understand how the system moves data up and down the memory hierarchy, then you can write your application programs so that their data items are stored higher in the hierarchy, where the CPU can access them more quickly.

This idea centers around a fundamental property of computer programs known as locality. Programs with good locality tend to access the same set of data items over and over again, or they tend to access sets of nearby data items. Programs with good locality tend to access more data items from the upper levels of the memory hierarchy than programs with poor locality, and thus run faster. For example, the running times of different matrix multiplication kernels that perform the same number of arithmetic operations, but have different degrees of locality, can vary by a factor of 20!

In this chapter, we will look at the basic storage technologies—SRAM memory, DRAM memory, ROM memory, and rotating and solid state disks—and describe how they are organized into hierarchies. In particular, we focus on the cache memories that act as staging areas between the CPU and main memory, because they have the most impact on application program performance. We show you how to analyze your C programs for locality and we introduce techniques for improving the locality in your programs. You will also learn an interesting way to characterize the performance of the memory hierarchy on a particular machine as a “memory mountain” that shows read access times as a function of locality.
6.1 Storage Technologies

Much of the success of computer technology stems from the tremendous progress in storage technology. Early computers had a few kilobytes of random-access memory. The earliest IBM PCs didn’t even have a hard disk. That changed with the introduction of the IBM PC-XT in 1982, with its 10-megabyte disk. By the year 2010, typical machines had 150,000 times as much disk storage, and the amount of storage was increasing by a factor of 2 every couple of years.

6.1.1 Random-Access Memory

Random-access memory (RAM) comes in two varieties—static and dynamic. Static RAM (SRAM) is faster and significantly more expensive than Dynamic RAM (DRAM). SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAM stores each bit in a bistable memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or states. Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable states. Such a memory cell is analogous to the inverted pendulum illustrated in Figure 6.1.

The pendulum is stable when it is tilted either all the way to the left or all the way to the right. From any other position, the pendulum will fall to one side or the other. In principle, the pendulum could also remain balanced in a vertical position indefinitely, but this state is metastable—the smallest disturbance would make it start to fall, and once it fell it would never return to the vertical position.

Due to its bistable nature, an SRAM memory cell will retain its value indefinitely, as long as it is kept powered. Even when a disturbance, such as electrical noise, perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.
### Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads, that is, $30 \times 10^{-15}$ farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense—each cell consists of a capacitor and a single access transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

Various sources of leakage current cause a DRAM cell to lose its charge within a time period of around 10 to 100 milliseconds. Fortunately, for computers operating with clock cycle times measured in nanoseconds, this retention time is quite long. The memory system must periodically refresh every bit of memory by reading it out and then rewriting it. Some systems also use error-correcting codes, where the computer words are encoded a few more bits (e.g., a 32-bit word might be encoded using 38 bits), such that circuitry can detect and correct any single erroneous bit within a word.

Figure 6.2 summarizes the characteristics of SRAM and DRAM memory. SRAM is persistent as long as power is applied. Unlike DRAM, no refresh is necessary. SRAM can be accessed faster than DRAM. SRAM is not sensitive to disturbances such as light and electrical noise. The trade-off is that SRAM cells use more transistors than DRAM cells, and thus have lower densities, are more expensive, and consume more power.

### Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into $d$ supercells, each consisting of $w$ DRAM cells. A $d \times w$ DRAM stores a total of $dw$ bits of information. The supercells are organized as a rectangular array with $r$ rows and $c$ columns, where $rc = d$. Each supercell has an address of the form $(i, j)$, where $i$ denotes the row, and $j$ denotes the column.

For example, Figure 6.3 shows the organization of a $16 \times 8$ DRAM chip with $d = 16$ supercells, $w = 8$ bits per supercell, $r = 4$ rows, and $c = 4$ columns. The shaded box denotes the supercell at address $(2, 1)$. Information flows in and out of the chip via external connectors called pins. Each pin carries a 1-bit signal. Figure 6.3 shows two of these sets of pins: eight data pins that can transfer 1 byte...
in or out of the chip, and two addr pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

**Aside** A note on terminology

The storage community has never settled on a standard name for a DRAM array element. Computer architects tend to refer to it as a “cell,” overloading the term with the DRAM storage cell. Circuit designers tend to refer to it as a “word,” overloading the term with a word of main memory. To avoid confusion, we have adopted the unambiguous term “supercell.”

Each DRAM chip is connected to some circuitry, known as the memory controller, that can transfer \( w \) bits at a time to and from each DRAM chip. To read the contents of supercell \((i, j)\), the memory controller sends the row address \( i \) to the DRAM, followed by the column address \( j \). The DRAM responds by sending the contents of supercell \((i, j)\) back to the controller. The row address \( i \) is called a **RAS (Row Access Strobe) request**. The column address \( j \) is called a **CAS (Column Access Strobe) request**. Notice that the RAS and CAS requests share the same DRAM address pins.

For example, to read supercell \((2, 1)\) from the \( 16 \times 8 \) DRAM in Figure 6.3, the memory controller sends row address 2, as shown in Figure 6.4(a). The DRAM responds by copying the entire contents of row 2 into an internal row buffer. Next, the memory controller sends column address 1, as shown in Figure 6.4(b). The DRAM responds by copying the 8 bits in supercell \((2, 1)\) from the row buffer and sending them to the memory controller.

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to reduce the number of address pins on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address pins instead of two. The disadvantage of the two-dimensional array organization is that addresses must be sent in two distinct steps, which increases the access time.
Chapter 6  The Memory Hierarchy

Memory controller

DRAM chip

(a) Select row 2 (RAS request).

(b) Select column 1 (CAS request).

Figure 6.4  Reading the contents of a DRAM supercell.

Memory Modules

DRAM chips are packaged in memory modules that plug into expansion slots on the main system board (motherboard). Common packages include the 168-pin dual inline memory module (DIMM), which transfers data to and from the memory controller in 64-bit chunks, and the 72-pin single inline memory module (SIMM), which transfers data in 32-bit chunks.

Figure 6.5 shows the basic idea of a memory module. The example module stores a total of 64 MB (megabytes) using eight 64-Mbit $8M \times 8$ DRAM chips, numbered 0 to 7. Each supercell stores 1 byte of main memory, and each 64-bit doubleword\footnote{IA32 would call this 64-bit quantity a “quadword.”} at byte address $A$ in main memory is represented by the eight supercells whose corresponding supercell address is $(i, j)$. In the example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

To retrieve a 64-bit doubleword at memory address $A$, the memory controller converts $A$ to a supercell address $(i, j)$ and sends it to the memory module, which then broadcasts $i$ and $j$ to each DRAM. In response, each DRAM outputs the 8-bit contents of its $(i, j)$ supercell. Circuitry in the module collects these outputs and forms them into a 64-bit doubleword, which it returns to the memory controller.

Main memory can be aggregated by connecting multiple memory modules to the memory controller. In this case, when the controller receives an address $A$, the controller selects the module $k$ that contains $A$, converts $A$ to its $(i, j)$ form, and sends $(i, j)$ to module $k$. 

1. IA32 would call this 64-bit quantity a “quadword.”
Section 6.1 Storage Technologies

Figure 6.5  Reading the contents of a memory module.

Practice Problem 6.1

In the following, let $r$ be the number of rows in a DRAM array, $c$ the number of columns, $b_r$ the number of bits needed to address the rows, and $b_c$ the number of bits needed to address the columns. For each of the following DRAMs, determine the power-of-two array dimensions that minimize $\max(b_r, b_c)$, the maximum number of bits needed to address the rows or columns of the array.

<table>
<thead>
<tr>
<th>Organization</th>
<th>$r$</th>
<th>$c$</th>
<th>$b_r$</th>
<th>$b_c$</th>
<th>$\max(b_r, b_c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$16 \times 4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$128 \times 8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$512 \times 4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1024 \times 4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing
processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

- **Fast page mode DRAM (FPM DRAM).** A conventional DRAM copies an entire row of supercells into its internal row buffer, uses one, and then discards the rest. FPM DRAM improves on this by allowing consecutive accesses to the same row to be served directly from the row buffer. For example, to read four supercells from row \( i \) of a conventional DRAM, the memory controller must send four RAS/CAS requests, even though the row address \( i \) is identical in each case. To read supercells from the same row of an FPM DRAM, the memory controller sends an initial RAS/CAS request, followed by three CAS requests. The initial RAS/CAS request copies row \( i \) into the row buffer and returns the supercell addressed by the CAS. The next three supercells are served directly from the row buffer, and thus more quickly than the initial supercell.

- **Extended data out DRAM (EDO DRAM).** An enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time.

- **Synchronous DRAM (SDRAM).** Conventional, FPM, and EDO DRAMs are asynchronous in the sense that they communicate with the memory controller using a set of explicit control signals. SDRAM replaces many of these control signals with the rising edges of the same external clock signal that drives the memory controller. Without going into detail, the net effect is that an SDRAM can output the contents of its supercells at a faster rate than its asynchronous counterparts.

- **Double Data-Rate Synchronous DRAM (DDR SDRAM).** DDR SDRAM is an enhancement of SDRAM that doubles the speed of the DRAM by using both clock edges as control signals. Different types of DDR SDRAMs are characterized by the size of a small prefetch buffer that increases the effective bandwidth: DDR (2 bits), DDR2 (4 bits), and DDR3 (8 bits).

- **Rambus DRAM (RDRAM).** This is an alternative proprietary technology with a higher maximum bandwidth than DDR SDRAM.

- **Video RAM (VRAM).** Used in the frame buffers of graphics systems. VRAM is similar in spirit to FPM DRAM. Two major differences are that (1) VRAM output is produced by shifting the entire contents of the internal buffer in sequence, and (2) VRAM allows concurrent reads and writes to the memory. Thus, the system can be painting the screen with the pixels in the frame buffer (reads) while concurrently writing new values for the next update (writes).

**Aside**  
**Historical popularity of DRAM technologies**

Until 1995, most PCs were built with FPM DRAMs. From 1996 to 1999, EDO DRAMs dominated the market, while FPM DRAMs all but disappeared. SDRAMs first appeared in 1995 in high-end systems, and by 2002 most PCs were built with SDRAMs and DDR SDRAMs. By 2010, most server and desktop systems were built with DDR3 SDRAMs. In fact, the Intel Core i7 supports only DDR3 SDRAM.
Nonvolatile Memory

DRAMs and SRAMs are volatile in the sense that they lose their information if the supply voltage is turned off. Nonvolatile memories, on the other hand, retain their information even when they are powered off. There are a variety of nonvolatile memories. For historical reasons, they are referred to collectively as read-only memories (ROMs), even though some types of ROMs can be written to as well as read. ROMs are distinguished by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them.

A programmable ROM (PROM) can be programmed exactly once. PROMs include a sort of fuse with each memory cell that can be blown once by zapping it with a high current.

An erasable programmable ROM (EPROM) has a transparent quartz window that permits light to reach the storage cells. The EPROM cells are cleared to zeros by shining ultraviolet light through the window. Programming an EPROM is done by using a special device to write ones into the EPROM. An EPROM can be erased and reprogrammed on the order of 1000 times. An electrically erasable PROM (EEPROM) is akin to an EPROM, but does not require a physically separate programming device, and thus can be reprogrammed in-place on printed circuit cards. An EEPROM can be reprogrammed on the order of $10^5$ times before it wears out.

Flash memory is a type of nonvolatile memory, based on EEPROMs, that has become an important storage technology. Flash memories are everywhere, providing fast and durable nonvolatile storage for a slew of electronic devices, including digital cameras, cell phones, music players, PDAs, and laptop, desktop, and server computer systems. In Section 6.1.3, we will look in detail at a new form of flash-based disk drive, known as a solid state disk (SSD), that provides a faster, sturdier, and less power-hungry alternative to conventional rotating disks.

Programs stored in ROM devices are often referred to as firmware. When a computer system is powered up, it runs firmware stored in a ROM. Some systems provide a small set of primitive input and output functions in firmware, for example, a PC’s BIOS (basic input/output system) routines. Complicated devices such as graphics cards and disk drive controllers also rely on firmware to translate I/O (input/output) requests from the CPU.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called buses. Each transfer of data between the CPU and memory is accomplished with a series of steps called a bus transaction. A read transaction transfers data from the main memory to the CPU. A write transaction transfers data from the CPU to the main memory.

A bus is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example,
is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of an example computer system. The main components are the CPU chip, a chipset that we will call an I/O bridge (which includes the memory controller), and the DRAM memory modules that make up main memory. These components are connected by a pair of buses: a system bus that connects the CPU to the I/O bridge, and a memory bus that connects the I/O bridge to the main memory.

The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

**Aside**  A note on bus designs

Bus design is a complex and rapidly changing aspect of computer systems. Different vendors develop different bus architectures as a way to differentiate their products. For example, Intel systems use chipsets known as the northbridge and the southbridge to connect the CPU to memory and I/O devices, respectively. In older Pentium and Core 2 systems, a front side bus (FSB) connects the CPU to the northbridge. Systems from AMD replace the FSB with the HyperTransport interconnect, while newer Intel Core i7 systems use the QuickPath interconnect. The details of these different bus architectures are beyond the scope of this text. Instead, we will use the high-level bus architecture from Figure 6.6 as a running example throughout the text. It is a simple but useful abstraction that allows us to be concrete, and captures the main ideas without being tied too closely to the detail of any proprietary designs.

Consider what happens when the CPU performs a load operation such as

```assembly
movl A, %eax
```

where the contents of address A are loaded into register %eax. Circuitry on the CPU chip called the bus interface initiates a read transaction on the bus. The read transaction consists of three steps. First, the CPU places the address A on the system bus. The I/O bridge passes the signal along to the memory bus (Figure 6.7(a)). Next, the main memory senses the address signal on the memory
Section 6.1 Storage Technologies

(a) CPU places address A on the memory bus.

(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.

(c) CPU reads word x from the bus, and copies it into register %eax.

Figure 6.7 **Memory read transaction for a load operation**: movl A, %eax.

... read transaction for a load operation: movl A, %eax.

... reads the address from the memory bus, fetches the data word from the DRAM, and writes the data to the memory bus. The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads it from the bus, and copies it to register %eax (Figure 6.7(c)).

Conversely, when the CPU performs a store instruction such as

```plaintext
movl %eax, A
```

where the contents of register %eax are written to address A, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data word in %eax to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).
6.1.2 Disk Storage

Disks are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.

Disk Geometry

Disks are constructed from platters. Each platter consists of two sides, or surfaces, that are coated with magnetic recording material. A rotating spindle in the center of the platter spins the platter at a fixed rotational rate, typically between 5400 and
15,000 revolutions per minute (RPM). A disk will typically contain one or more of these platters encased in a sealed container.

Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called tracks. Each track is partitioned into a collection of sectors. Each sector contains an equal number of data bits (typically 512 bytes) encoded in the magnetic material on the sector. Sectors are separated by gaps where no data bits are stored. Gaps store formatting bits that identify sectors.

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a disk drive, although we will usually refer to it as simply a disk. We will sometime refer to disks as rotating disks to distinguish them from flash-based solid state disks (SSDs), which have no moving parts.

Disk manufacturers describe the geometry of multiple-platter drives in terms of cylinders, where a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. For example, if a drive has three platters and six surfaces, and the tracks on each surface are numbered consistently, then cylinder $k$ is the collection of the six instances of track $k$.

**Disk Capacity**

The maximum number of bits that can be recorded by a disk is known as its maximum capacity, or simply capacity. Disk capacity is determined by the following technology factors:

- **Recording density (bits/in):** The number of bits that can be squeezed into a 1-inch segment of a track.
- **Track density (tracks/in):** The number of tracks that can be squeezed into a 1-inch segment of the radius extending from the center of the platter.
• Areal density (bits/in²): The product of the recording density and the track density.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every few years. The original disks, designed in an age of low areal density, partitioned every track into the same number of sectors, which was determined by the number of sectors that could be recorded on the innermost track. To maintain a fixed number of sectors per track, the sectors were spaced farther apart on the outer tracks. This was a reasonable approach when areal densities were relatively low. However, as areal densities increased, the gaps between sectors (where no data bits were stored) became unacceptably large. Thus, modern high-capacity disks use a technique known as multiple zone recording, where the set of cylinders is partitioned into disjoint subsets known as recording zones. Each zone consists of a contiguous collection of cylinders. Each track in each cylinder in zone has the same number of sectors, which is determined by the number of sectors that can be packed into the innermost track of the zone. Note that diskettes (floppy disks) still use the old-fashioned approach, with a constant number of sectors per track.

The capacity of a disk is given by the following formula:

\[
\text{Disk capacity} = \frac{\# \text{ bytes sector}}{\text{sector}} \times \frac{\text{average \# sectors track}}{\text{track}} \times \frac{\text{\# tracks surface}}{\text{surface}} \times \frac{\text{\# surfaces platter}}{\text{platter}} \times \frac{\text{\# platters disk}}{\text{disk}}
\]

For example, suppose we have a disk with 5 platters, 512 bytes per sector, 20,000 tracks per surface, and an average of 300 sectors per track. Then the capacity of the disk is:

\[
\text{Disk capacity} = \frac{512 \text{ bytes sector}}{\text{sector}} \times \frac{300 \text{ sectors track}}{\text{track}} \times \frac{20,000 \text{ tracks surface}}{\text{surface}} \times \frac{2 \text{ surfaces platter}}{\text{platter}} \times \frac{5 \text{ platters disk}}{\text{disk}}
\]

\[
= 30,720,000,000 \text{ bytes}
\]

\[
= 30.72 \text{ GB}.
\]

Notice that manufacturers express disk capacity in units of gigabytes (GB), where 1 GB = 10⁹ bytes.

---

**Aside** How much is a gigabyte?

Unfortunately, the meanings of prefixes such as kilo (K), mega (M), giga (G), and tera (T) depend on the context. For measures that relate to the capacity of DRAMs and SRAMs, typically \( K = 2^{10} \), \( M = 2^{20} \), \( G = 2^{30} \), and \( T = 2^{40} \). For measures related to the capacity of I/O devices such as disks and networks, typically \( K = 10^3 \), \( M = 10^6 \), \( G = 10^9 \), and \( T = 10^{12} \). Rates and throughputs usually use these prefix values as well.

Fortunately, for the back-of-the-envelope estimates that we typically rely on, either assumption works fine in practice. For example, the relative difference between \( 2^{20} = 1,048,576 \) and \( 10^6 = 1,000,000 \) is small: \( (2^{20} - 10^6)/10^6 \approx 5\% \). Similarly for \( 2^{30} = 1,073,741,824 \) and \( 10^9 = 1,000,000,000: (2^{30} - 10^9)/10^9 \approx 7\% \).
Section 6.1 Storage Technologies

Spindle

The disk surface spins at a fixed rotational rate. The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air. By moving radially, the arm can position the read/write head over any track.

(a) Single-platter view

Figure 6.10 Disk dynamics.

(b) Multiple-platter view

Practice Problem 6.2

What is the capacity of a disk with two platters, 10,000 cylinders, an average of 400 sectors per track, and 512 bytes per sector?

Disk Operation

Disks read and write bits stored on the magnetic surface using a read/write head connected to the end of an actuator arm, as shown in Figure 6.10(a). By moving the arm back and forth along its radial axis, the drive can position the head over any track on the surface. This mechanical motion is known as a seek. Once the head is positioned over the desired track, then as each bit on the track passes underneath, the head can either sense the value of the bit (read the bit) or alter the value of the bit (write the bit). Disks with multiple platters have a separate read/write head for each surface, as shown in Figure 6.10(b). The heads are lined up vertically and move in unison. At any point in time, all heads are positioned on the same cylinder.

The read/write head at the end of the arm flies (literally) on a thin cushion of air over the disk surface at a height of about 0.1 microns and a speed of about 80 km/h. This is analogous to placing the Sears Tower on its side and flying it around the world at a height of 2.5 cm (1 inch) above the ground, with each orbit of the earth taking only 8 seconds! At these tolerances, a tiny piece of dust on the surface is like a huge boulder. If the head were to strike one of these boulders, the head would cease flying and crash into the surface (a so-called head crash). For this reason, disks are always sealed in airtight packages.

Disks read and write data in sector-sized blocks. The access time for a sector has three main components: seek time, rotational latency, and transfer time:
• **Seek time:** To read the contents of some target sector, the arm first positions the head over the track that contains the target sector. The time required to move the arm is called the *seek time*. The seek time, $T_{\text{seek}}$, depends on the previous position of the head and the speed that the arm moves across the surface. The average seek time in modern drives, $T_{\text{avgseek}}$, measured by taking the mean of several thousand seeks to random sectors, is typically on the order of 3 to 9 ms. The maximum time for a single seek, $T_{\text{maxseek}}$, can be as high as 20 ms.

• **Rotational latency:** Once the head is in position over the track, the drive waits for the first bit of the target sector to pass under the head. The performance of this step depends on both the position of the surface when the head arrives at the target sector and the rotational speed of the disk. In the worst case, the head just misses the target sector and waits for the disk to make a full rotation. Thus, the maximum rotational latency, in seconds, is given by

$$T_{\text{maxrotation}} = \frac{1}{\text{RPM}} \times \frac{60 \text{ secs}}{1 \text{ min}}$$

The average rotational latency, $T_{\text{avgrotation}}$, is simply half of $T_{\text{maxrotation}}$.

• **Transfer time:** When the first bit of the target sector is under the head, the drive can begin to read or write the contents of the sector. The transfer time for one sector depends on the rotational speed and the number of sectors per track. Thus, we can roughly estimate the average transfer time for one sector in seconds as

$$T_{\text{avgtransfer}} = \frac{1}{\text{RPM}} \times \frac{1}{\text{(average # sectors/track)}} \times \frac{60 \text{ secs}}{1 \text{ min}}$$

We can estimate the average time to access the contents of a disk sector as the sum of the average seek time, the average rotational latency, and the average transfer time. For example, consider a disk with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotational rate</td>
<td>7200 RPM</td>
</tr>
<tr>
<td>$T_{\text{avgseek}}$</td>
<td>9 ms</td>
</tr>
<tr>
<td>Average # sectors/track</td>
<td>400</td>
</tr>
</tbody>
</table>

For this disk, the average rotational latency (in ms) is

$$T_{\text{avgrotation}} = \frac{1}{2} \times T_{\text{maxrotation}}$$

$$= \frac{1}{2} \times \left(\frac{60 \text{ secs}}{7200 \text{ RPM}}\right) \times 1000 \text{ ms/sec}$$

$$\approx 4 \text{ ms}$$

The average transfer time is

$$T_{\text{avgtransfer}} = \frac{60}{7200 \text{ RPM}} \times \frac{1}{400 \text{ sectors/track}} \times 1000 \text{ ms/sec}$$

$$\approx 0.02 \text{ ms}$$
Putting it all together, the total estimated access time is

\[ T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} \]

\[ = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms} \]

\[ = 13.02 \text{ ms} \]

This example illustrates some important points:

- The time to access the 512 bytes in a disk sector is dominated by the seek time and the rotational latency. Accessing the first byte in the sector takes a long time, but the remaining bytes are essentially free.
- Since the seek time and rotational latency are roughly the same, twice the seek time is a simple and reasonable rule for estimating disk access time.
- The access time for a doubleword stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-sized block from memory is roughly 256 ns for SRAM and 4000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2500 times greater than DRAM. The difference in access times is even more dramatic if we compare the times to access a single word.

\[ \text{Practice Problem 6.3} \]

Estimate the average time (in ms) to access a sector on the following disk:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotational rate</td>
<td>15,000 RPM</td>
</tr>
<tr>
<td>( T_{\text{avg seek}} )</td>
<td>8 ms</td>
</tr>
<tr>
<td>Average # sectors/track</td>
<td>500</td>
</tr>
</tbody>
</table>

**Logical Disk Blocks**

As we have seen, modern disks have complex geometries, with multiple surfaces and different recording zones on those surfaces. To hide this complexity from the operating system, modern disks present a simpler view of their geometry as a sequence of \( B \) sector-sized logical blocks, numbered 0, 1, \ldots, \( B - 1 \). A small hardware/firmware device in the disk package, called the disk controller, maintains the mapping between logical block numbers and actual (physical) disk sectors.

When the operating system wants to perform an I/O operation such as reading a disk sector into main memory, it sends a command to the disk controller asking it to read a particular logical block number. Firmware on the controller performs a fast table lookup that translates the logical block number into a \((\text{surface}, \text{track}, \text{sector})\) triple that uniquely identifies the corresponding physical sector. Hardware on the controller interprets this triple to move the heads to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed
by the head into a small memory buffer on the controller, and copies them into main memory.

**Aside**  Formatted disk capacity

Before a disk can be used to store data, it must be *formatted* by the disk controller. This involves filling in the gaps between sectors with information that identifies the sectors, identifying any cylinders with surface defects and taking them out of action, and setting aside a set of cylinders in each zone as spares that can be called into action if one or more cylinders in the zone goes bad during the lifetime of the disk. The *formatted capacity* quoted by disk manufacturers is less than the maximum capacity because of the existence of these spare cylinders.

**Practice Problem 6.4**

Suppose that a 1 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotational rate</td>
<td>10,000 RPM</td>
</tr>
<tr>
<td>( T_{avg\ seek} )</td>
<td>5 ms</td>
</tr>
<tr>
<td>Average # sectors/track</td>
<td>1000</td>
</tr>
<tr>
<td>Surfaces</td>
<td>4</td>
</tr>
<tr>
<td>Sector size</td>
<td>512 bytes</td>
</tr>
</tbody>
</table>

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is \( T_{avg\ seek} + T_{avg\ rotation} \).

A. **Best case**: Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential).

B. **Random case**: Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

**Connecting I/O Devices**

Input/output (I/O) devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an *I/O bus* such as Intel’s *Peripheral Component Interconnect* (PCI) bus. Unlike the system bus and memory buses, which are CPU-specific, I/O buses such as PCI are designed to be independent of the underlying CPU. For example, PCs and Macs both incorporate the PCI bus. Figure 6.11 shows a typical I/O bus structure (modeled on PCI) that connects the CPU, main memory, and I/O devices.

Although the I/O bus is slower than the system and memory buses, it can accommodate a wide variety of third-party I/O devices. For example, the bus in Figure 6.11 has three different types of devices attached to it.
A Universal Serial Bus (USB) controller is a conduit for devices attached to a USB bus, which is a wildly popular standard for connecting a variety of peripheral I/O devices, including keyboards, mice, modems, digital cameras, game controllers, printers, external disk drives, and solid state disks. USB 2.0 buses have a maximum bandwidth of 60 MB/s. USB 3.0 buses have a maximum bandwidth of 600 MB/s.

A graphics card (or adapter) contains hardware and software logic that is responsible for painting the pixels on the display monitor on behalf of the CPU.

A host bus adapter that connects one or more disks to the I/O bus using a communication protocol defined by a particular host bus interface. The two most popular such interfaces for disks are SCSI (pronounced “scuzzy”) and SATA (pronounced “sat-uh”). SCSI disks are typically faster and more expensive than SATA drives. A SCSI host bus adapter (often called a SCSI controller) can support multiple disk drives, as opposed to SATA adapters, which can only support one drive.

Additional devices such as network adapters can be attached to the I/O bus by plugging the adapter into empty expansion slots on the motherboard that provide a direct electrical connection to the bus.
Chapter 6  The Memory Hierarchy

(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.

(b) The disk controller reads the sector and performs a DMA transfer into main memory.

Figure 6.12  Reading a disk sector.

Accessing Disks

While a detailed description of how I/O devices work and how they are programmed is outside our scope here, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.

The CPU issues commands to I/O devices using a technique called memory-mapped I/O (Figure 6.12(a)). In a system with memory-mapped I/O, a block of
addresses in the address space is reserved for communicating with I/O devices. Each of these addresses is known as an I/O port. Each device is associated with (or mapped to) one or more ports when it is attached to the bus.

As a simple example, suppose that the disk controller is mapped to port 0xa0. Then the CPU might initiate a disk read by executing three store instructions to address 0xa0: The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished. (We will discuss interrupts in Section 8.1.) The second instruction indicates the logical block number that should be read. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is taking place would be enormously wasteful.

After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU (Figure 6.12(b)). This process, whereby a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as direct memory access (DMA). The transfer of data is known as a DMA transfer.

After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is
Anatomy of a Commercial Disk

Disk manufacturers publish a lot of useful high-level technical information on their Web pages. For example, the Cheetah 15K.4 is a SCSI disk first manufactured by Seagate in 2005. If we consult the online product manual on the Seagate Web page, we can glean the geometry and performance information shown in Figure 6.13.

Disk manufacturers rarely publish detailed technical information about the geometry of the individual recording zones. However, storage researchers at Carnegie Mellon University have developed a useful tool, called DIXtrac, that automatically discovers a wealth of low-level information about the geometry and performance of SCSI disks [92]. For example, DIXtrac is able to discover the detailed zone geometry of our example Seagate disk, which we’ve shown in Figure 6.14. Each row in the table characterizes one of the 15 zones. The first column gives the zone number, with zone 0 being the outermost and zone 14 the innermost. The second column gives the number of sectors contained in each track in that zone. The third column shows the number of cylinders assigned to that zone, where each cylinder consists of eight tracks, one from each surface. Similarly, the fourth column gives the total number of logical blocks assigned to each zone, across all eight surfaces. (The tool was not able to extract valid data for the innermost zone, so these are omitted.)

The zone map reveals some interesting facts about the Seagate disk. First, more sectors are packed into the outer zones (which have a larger circumference) than the inner zones. Second, each zone has more sectors than logical blocks.
Section 6.1 Storage Technologies

<table>
<thead>
<tr>
<th>Zone number</th>
<th>Sectors per track</th>
<th>Cylinders per zone</th>
<th>Logical blocks per zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>(outer) 0</td>
<td>864</td>
<td>3201</td>
<td>22,076,928</td>
</tr>
<tr>
<td>1</td>
<td>844</td>
<td>3200</td>
<td>21,559,136</td>
</tr>
<tr>
<td>2</td>
<td>816</td>
<td>3400</td>
<td>22,149,504</td>
</tr>
<tr>
<td>3</td>
<td>806</td>
<td>3100</td>
<td>19,943,664</td>
</tr>
<tr>
<td>4</td>
<td>795</td>
<td>3100</td>
<td>19,671,480</td>
</tr>
<tr>
<td>5</td>
<td>768</td>
<td>3400</td>
<td>20,852,736</td>
</tr>
<tr>
<td>6</td>
<td>768</td>
<td>3450</td>
<td>21,159,936</td>
</tr>
<tr>
<td>7</td>
<td>725</td>
<td>3650</td>
<td>21,135,200</td>
</tr>
<tr>
<td>8</td>
<td>704</td>
<td>3700</td>
<td>20,804,608</td>
</tr>
<tr>
<td>9</td>
<td>672</td>
<td>3700</td>
<td>19,858,944</td>
</tr>
<tr>
<td>10</td>
<td>640</td>
<td>3700</td>
<td>18,913,280</td>
</tr>
<tr>
<td>11</td>
<td>603</td>
<td>3700</td>
<td>17,819,856</td>
</tr>
<tr>
<td>12</td>
<td>576</td>
<td>3707</td>
<td>17,054,208</td>
</tr>
<tr>
<td>13</td>
<td>528</td>
<td>3060</td>
<td>12,900,096</td>
</tr>
<tr>
<td>(inner) 14</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 6.14 Seagate Cheetah 15K.4 zone map. Source: DIXtrac automatic disk drive characterization tool [92]. Data for zone 14 not available.

(check this yourself). These spare sectors form a pool of spare cylinders. If the recording material on a sector goes bad, the disk controller will automatically remap the logical blocks on that cylinder to an available spare. So we see that the notion of a logical block not only provides a simpler interface to the operating system, it also provides a level of indirection that enables the disk to be more robust. This general idea of indirection is very powerful, as we will see when we study virtual memory in Chapter 9.

**Practice Problem 6.5**

Use the zone map in Figure 6.14 to determine the number of spare cylinders in the following zones:

A. Zone 0
B. Zone 8

6.1.3 Solid State Disks

A solid state disk (SSD) is a storage technology, based on flash memory (Section 6.1.1), that in some situations is an attractive alternative to the conventional rotating disk. Figure 6.15 shows the basic idea. An SSD package plugs into a standard disk slot on the I/O bus (typically USB or SATA) and behaves like any other
disk, processing requests from the CPU to read and write logical disk blocks. An SSD package consists of one or more flash memory chips, which replace the mechanical drive in a conventional rotating disk, and a flash translation layer, which is a hardware/firmware device that plays the same role as a disk controller, translating requests for logical blocks into accesses of the underlying physical device.

SSDs have different performance characteristics than rotating disks. As shown in Figure 6.16, sequential reads and writes (where the CPU accesses logical disk blocks in sequential order) have comparable performance, with sequential reading somewhat faster than sequential writing. However, when logical blocks are accessed in random order, writing is an order of magnitude slower than reading.

The difference between random reading and writing performance is caused by a fundamental property of the underlying flash memory. As shown in Figure 6.15, a flash memory consists of a sequence of $B$ blocks, where each block consists of $P$ pages. Typically, pages are 512–4KB in size, and a block consists of 32–128 pages, with total block sizes ranging from 16 KB to 512 KB. Data is read and written in units of pages. A page can be written only after the entire block to which it belongs has been erased (typically this means that all bits in the block are set to 1). However, once a block is erased, each page in the block can be written once with no further erasing. A block wears out after roughly 100,000 repeated writes. Once a block wears out it can no longer be used.
Random writes are slow for two reasons. First, erasing a block takes a relatively long time, on the order of 1 ms, which is more than an order of magnitude longer than it takes to access a page. Second, if a write operation attempts to modify a page \( p \) that contains existing data (i.e., not all ones), then any pages in the same block with useful data must be copied to a new (erased) block before the write to page \( p \) can occur. Manufacturers have developed sophisticated logic in the flash translation layer that attempts to amortize the high cost of erasing blocks and to minimize the number of internal copies on writes, but it is unlikely that random writing will ever perform as well as reading.

SSDs have a number of advantages over rotating disks. They are built of semiconductor memory, with no moving parts, and thus have much faster random access times than rotating disks, use less power, and are more rugged. However, there are some disadvantages. First, because flash blocks wear out after repeated writes, SSDs have the potential to wear out as well. Wear leveling logic in the flash translation layer attempts to maximize the lifetime of each block by spreading erasures evenly across all blocks, but the fundamental limit remains. Second, SSDs are about 100 times more expensive per byte than rotating disks, and thus the typical storage capacities are 100 times less than rotating disks. However, SSD prices are decreasing rapidly as they become more popular, and the gap between the two appears to be decreasing.

SSDs have completely replaced rotating disks in portable music devices, are popular as disk replacements in laptops, and have even begun to appear in desktops and servers. While rotating disks are here to stay, it is clear that SSDs are an important new storage technology.

**Practice Problem 6.6**

As we have seen, a potential drawback of SSDs is that the underlying flash memory can wear out. For example, one major manufacturer guarantees 1 petabyte (\(10^{15}\) bytes) of random writes for their SSDs before they wear out. Given this assumption, estimate the lifetime (in years) of the SSD in Figure 6.16 for the following workloads:

A. **Worst case for sequential writes:** The SSD is written to continuously at a rate of 170 MB/s (the average sequential write throughput of the device).

B. **Worst case for random writes:** The SSD is written to continuously at a rate of 14 MB/s (the average random write throughput of the device).

C. **Average case:** The SSD is written to at a rate of 20 GB/day (the average daily write rate assumed by some computer manufacturers in their mobile computer workload simulations).

---

**6.1.4 Storage Technology Trends**

There are several important concepts to take away from our discussion of storage technologies.
Different storage technologies have different price and performance trade-offs. SRAM is somewhat faster than DRAM, and DRAM is much faster than disk. On the other hand, fast storage is always more expensive than slower storage. SRAM costs more per byte than DRAM. DRAM costs much more than disk. SSDs split the difference between DRAM and rotating disk.

The price and performance properties of different storage technologies are changing at dramatically different rates. Figure 6.17 summarizes the price and performance properties of storage technologies since 1980, when the first PCs were introduced. The numbers were culled from back issues of trade magazines and the Web. Although they were collected in an informal survey, the numbers reveal some interesting trends.

Since 1980, both the cost and performance of SRAM technology have improved at roughly the same rate. Access times have decreased by a factor of about 200 and cost per megabyte by a factor of 300 (Figure 6.17(a)). However, the trends
for DRAM and disk are much more dramatic and divergent. While the cost per megabyte of DRAM has decreased by a factor of 130,000 (more than five orders of magnitude!), DRAM access times have decreased by only a factor of 10 or so (Figure 6.17(b)). Disk technology has followed the same trend as DRAM and in even more dramatic fashion. While the cost of a megabyte of disk storage has plummeted by a factor of more than 1,000,000 (more than six orders of magnitude!) since 1980, access times have improved much more slowly, by only a factor of 30 or so (Figure 6.17(c)). These startling long-term trends highlight a basic truth of memory and disk technology: it is easier to increase density (and thereby reduce cost) than to decrease access time.

**DRAM and disk performance are lagging behind CPU performance.** As we see in Figure 6.17(d), CPU cycle times improved by a factor of 2500 between 1980 and 2010. If we look at the effective cycle time—which we define to be the cycle time of an individual CPU (processor) divided by the number of its processor cores—then the improvement between 1980 and 2010 is even greater, a factor of 10,000. The split in the CPU performance curve around 2003 reflects the introduction of multicore processors (see aside on next page). After this split, cycle times of individual cores actually increased a bit before starting to decrease again, albeit at a slower rate than before.

Note that while SRAM performance lags, it is roughly keeping up. However, the gap between DRAM and disk performance and CPU performance is actually widening. Until the advent of multi-core processors around 2003, this performance gap was a function of latency, with DRAM and disk access times increasing more slowly than the cycle time of an individual processor. However, with the introduction of multiple cores, this performance gap is increasingly a function of throughput, with multiple processor cores issuing requests to the DRAM and disk in parallel.

The various trends are shown quite clearly in Figure 6.18, which plots the access and cycle times from Figure 6.17 on a semi-log scale.

---

**Figure 6.18** The increasing gap between disk, DRAM, and CPU speeds.
As we will see in Section 6.4, modern computers make heavy use of SRAM-based caches to try to bridge the processor-memory gap. This approach works because of a fundamental property of application programs known as locality, which we discuss next.

**Aside**  When cycle time stood still: the advent of multi-core processors

The history of computers is marked by some singular events that caused profound changes in the industry and the world. Interestingly, these inflection points tend to occur about once per decade: the development of Fortran in the 1950s, the introduction of the IBM 360 in the early 1960s, the dawn of the Internet (then called ARPANET) in the early 1970s, the introduction of the IBM PC in the early 1980s, and the creation of the World Wide Web in the early 1990s.

The most recent such event occurred early in the 21st century, when computer manufacturers ran headlong into the so-called “power wall,” discovering that they could no longer increase CPU clock frequencies as quickly because the chips would then consume too much power. The solution was to improve performance by replacing a single large processor with multiple smaller processor cores, each a complete processor capable of executing programs independently and in parallel with the other cores. This multi-core approach works in part because the power consumed by a processor is proportional to $P = fCv^2$, where $f$ is the clock frequency, $C$ is the capacitance, and $v$ is the voltage. The capacitance $C$ is roughly proportional to the area, so the power drawn by multiple cores can be held constant as long as the total area of the cores is constant. As long as feature sizes continue to shrink at the exponential Moore’s law rate, the number of cores in each processor, and thus its effective performance, will continue to increase.

From this point forward, computers will get faster not because the clock frequency increases, but because the number of cores in each processor increases, and because architectural innovations increase the efficiency of programs running on those cores. We can see this trend clearly in Figure 6.18. CPU cycle time reached its lowest point in 2003 and then actually started to rise before leveling off and starting to decline again at a slower rate than before. However, because of the advent of multi-core processors (dual-core in 2004 and quad-core in 2007), the effective cycle time continues to decrease at close to its previous rate.

**Practice Problem 6.7**

Using the data from the years 2000 to 2010 in Figure 6.17(c), estimate the year when you will be able to buy a petabyte ($10^{15}$ bytes) of rotating disk storage for $500. Assume constant dollars (no inflation).

**6.2 Locality**

Well-written computer programs tend to exhibit good locality. That is, they tend to reference data items that are near other recently referenced data items, or that were recently referenced themselves. This tendency, known as the principle of locality, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.
Locality is typically described as having two distinct forms: **temporal locality** and **spatial locality**. In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Programmers should understand the principle of locality because, in general, **programs with good locality run faster than programs with poor locality**. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality. At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as cache memories that hold blocks of the most recently referenced instructions and data items. At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system. The principle of locality also plays a crucial role in the design of application programs. For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk. High-volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

### 6.2.1 Locality of References to Program Data

Consider the simple function in Figure 6.19(a) that sums the elements of a vector. Does this function have good locality? To answer this question, we look at the reference pattern for each variable. In this example, the `sum` variable is referenced once in each loop iteration, and thus there is good temporal locality with respect to `sum`. On the other hand, since `sum` is a scalar, there is no spatial locality with respect to `sum`.

As we see in Figure 6.19(b), the elements of vector `v` are read sequentially, one after the other, in the order they are stored in memory (we assume for convenience that the array starts at address 0). Thus, with respect to variable `v`, the function has good spatial locality but poor temporal locality since each vector element

```
int sumvec(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
}
```

*Figure 6.19  (a) A function with good locality. (b) Reference pattern for vector `v` (N = 8). Notice how the vector elements are accessed in the same order that they are stored in memory.*
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}

Figure 6.20 (a) Another function with good locality. (b) Reference pattern for array \texttt{a} (M = 2, N = 3).

There is good spatial locality because the array is accessed in the same row-major order in which it is stored in memory.

is accessed exactly once. Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the \texttt{sumvec} function enjoys good locality.

A function such as \texttt{sumvec} that visits each element of a vector sequentially is said to have a \textit{stride-1 reference pattern} (with respect to the element size). We will sometimes refer to stride-1 reference patterns as \textit{sequential reference patterns}. Visiting every \(k\)th element of a contiguous vector is called a \textit{stride-\(k\) reference pattern}. Stride-1 reference patterns are a common and important source of spatial locality in programs. In general, as the stride increases, the spatial locality decreases.

Stride is also an important issue for programs that reference multidimensional arrays. For example, consider the \texttt{sumarrayrows} function in Figure 6.20(a) that sums the elements of a two-dimensional array. The doubly nested loop reads the elements of the array in \textit{row-major order}. That is, the inner loop reads the elements of the first row, then the second row, and so on. The \texttt{sumarrayrows} function enjoys good spatial locality because it references the array in the same row-major order that the array is stored (Figure 6.20(b)). The result is a nice stride-1 reference pattern with excellent spatial locality.

Seemingly trivial changes to a program can have a big impact on its locality. For example, the \texttt{sumarraycols} function in Figure 6.21(a) computes the same result as the \texttt{sumarrayrows} function in Figure 6.20(a). The only difference is that we have interchanged the \(i\) and \(j\) loops. What impact does interchanging the loops have on its locality? The \texttt{sumarraycols} function suffers from poor spatial locality because it scans the array column-wise instead of row-wise. Since C arrays are laid out in memory row-wise, the result is a stride-\(N\) reference pattern, as shown in Figure 6.21(b).

### 6.2.2 Locality of Instruction Fetches

Since program instructions are stored in memory and must be fetched (read) by the CPU, we can also evaluate the locality of a program with respect to its instruction fetches. For example, in Figure 6.19 the instructions in the body of the
```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

**Figure 6.21** (a) A function with poor spatial locality. (b) Reference pattern for array \(a\) \((M = 2, N = 3)\).

The function has poor spatial locality because it scans memory with a stride-\(N\) reference pattern.

...for loop are executed in sequential memory order, and thus the loop enjoys good spatial locality. Since the loop body is executed multiple times, it also enjoys good temporal locality.

An important property of code that distinguishes it from program data is that it is rarely modified at run time. While a program is executing, the CPU reads its instructions from memory. The CPU rarely overwrites or modifies these instructions.

### 6.2.3 Summary of Locality

In this section, we have introduced the fundamental idea of locality and have identified some simple rules for qualitatively evaluating the locality in a program:

- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-\(k\) reference patterns, the smaller the stride the better the spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.
- Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.

Later in this chapter, after we have learned about cache memories and how they work, we will show you how to quantify the idea of locality in terms of cache hits and misses. It will also become clear to you why programs with good locality typically run faster than programs with poor locality. Nonetheless, knowing how to
glance at a source code and getting a high-level feel for the locality in the program is a useful and important skill for a programmer to master.

**Practice Problem 6.8**

Permute the loops in the following function so that it scans the three-dimensional array `a` with a stride-1 reference pattern.

```c
int sumarray3d(int a[N][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

**Practice Problem 6.9**

The three functions in Figure 6.22 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

(a) An array of structs

```c
#define N 1000

typedef struct {
    int vel[3];
    int acc[3];
} point;

point p[N];
```

(b) The `clear1` function

```c
void clear1(point *p, int n)
{
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++)
            p[i].vel[j] = 0;
        for (j = 0; j < 3; j++)
            p[i].acc[j] = 0;
    }
}
```

*Figure 6.22 Code examples for Practice Problem 6.9.*
6.3 The Memory Hierarchy

Sections 6.1 and 6.2 described some fundamental and enduring properties of storage technology and computer software:

- **Storage technology**: Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.
- **Computer software**: Well-written programs tend to exhibit good locality.

In one of the happier coincidences of computing, these fundamental properties of hardware and software complement each other beautifully. Their complementary nature suggests an approach for organizing memory systems, known as the **memory hierarchy**, that is used in all modern computer systems. Figure 6.23 shows a typical memory hierarchy. In general, the storage devices get slower, cheaper, and larger as we move from higher to lower levels. At the highest level (L0) are a small number of fast CPU registers that the CPU can access in a single clock cycle. Next are one or more small to moderate-sized SRAM-based cache memories that can be accessed in a few CPU clock cycles. These are followed by a large DRAM-based main memory that can be accessed in tens to hundreds of clock cycles. Next are slow but enormous local disks. Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network. For example, distributed file systems such as the Andrew File System (AFS) or the Network File System (NFS) allow a program to access files that are stored on remote network-connected servers. Similarly, the World Wide Web allows programs to access remote files stored on Web servers anywhere in the world.
Chapter 6  The Memory Hierarchy

CPU registers hold words retrieved from cache memory.

L1 cache holds cache lines retrieved from L2 cache.

L2 cache holds cache lines retrieved from L3 cache.

L3 cache holds cache lines retrieved from memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote network servers.

Remote secondary storage (distributed file systems, Web servers)

Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

Figure 6.23  The memory hierarchy.

Aside  Other memory hierarchies

We have shown you one example of a memory hierarchy, but other combinations are possible, and indeed common. For example, many sites back up local disks onto archival magnetic tapes. At some of these sites, human operators manually mount the tapes onto tape drives as needed. At other sites, tape robots handle this task automatically. In either case, the collection of tapes represents a level in the memory hierarchy, below the local disk level, and the same general principles apply. Tapes are cheaper per byte than disks, which allows sites to archive multiple snapshots of their local disks. The trade-off is that tapes take longer to access than disks. As another example, solid state disks are playing an increasingly important role in the memory hierarchy, bridging the gulf between DRAM and rotating disk.

6.3.1 Caching in the Memory Hierarchy

In general, a cache (pronounced “cash”) is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device. The process of using a cache is known as caching (pronounced “cashing”).

The central idea of a memory hierarchy is that for each $k$, the faster and smaller storage device at level $k$ serves as a cache for the larger and slower storage device at level $k + 1$. In other words, each level in the hierarchy caches data objects from the next lower level. For example, the local disk serves as a cache for files (such as Web pages) retrieved from remote disks over the network, the main memory serves as a cache for data on the local disks, and so on, until we get to the smallest cache of all, the set of CPU registers.
Section 6.3 The Memory Hierarchy

Figure 6.24  The basic principle of caching in a memory hierarchy.

Figure 6.24 shows the general concept of caching in a memory hierarchy. The storage at level \( k + 1 \) is partitioned into contiguous chunks of data objects called *blocks*. Each block has a unique address or name that distinguishes it from other blocks. Blocks can be either fixed-sized (the usual case) or variable-sized (e.g., the remote HTML files stored on Web servers). For example, the level \( k + 1 \) storage in Figure 6.24 is partitioned into 16 fixed-sized blocks, numbered 0 to 15.

Similarly, the storage at level \( k \) is partitioned into a smaller set of blocks that are the same size as the blocks at level \( k + 1 \). At any point in time, the cache at level \( k \) contains copies of a subset of the blocks from level \( k + 1 \). For example, in Figure 6.24, the cache at level \( k \) has room for four blocks and currently contains copies of blocks 4, 9, 14, and 3.

Data is always copied back and forth between level \( k \) and level \( k + 1 \) in block-sized transfer units. It is important to realize that while the block size is fixed between any particular pair of adjacent levels in the hierarchy, other pairs of levels can have different block sizes. For example, in Figure 6.23, transfers between L1 and L0 typically use one-word blocks. Transfers between L2 and L1 (and L3 and L2, and L4 and L3) typically use blocks of 8 to 16 words. And transfers between L5 and L4 use blocks with hundreds or thousands of bytes. In general, devices lower in the hierarchy (further from the CPU) have longer access times, and thus tend to use larger block sizes in order to amortize these longer access times.

**Cache Hits**

When a program needs a particular data object \( d \) from level \( k + 1 \), it first looks for \( d \) in one of the blocks currently stored at level \( k \). If \( d \) happens to be cached at level \( k \), then we have what is called a *cache hit*. The program reads \( d \) directly from level \( k \), which by the nature of the memory hierarchy is faster than reading \( d \) from level \( k + 1 \). For example, a program with good temporal locality might read a data object from block 14, resulting in a cache hit from level \( k \).
Cache Misses

If, on the other hand, the data object \( d \) is not cached at level \( k \), then we have what is called a cache miss. When there is a miss, the cache at level \( k \) fetches the block containing \( d \) from the cache at level \( k + 1 \), possibly overwriting an existing block if the level \( k \) cache is already full.

This process of overwriting an existing block is known as replacing or evicting the block. The block that is evicted is sometimes referred to as a victim block. The decision about which block to replace is governed by the cache’s replacement policy. For example, a cache with a random replacement policy would choose a random victim block. A cache with a least-recently used (LRU) replacement policy would choose the block that was last accessed the furthest in the past.

After the cache at level \( k \) has fetched the block from level \( k + 1 \), the program can read \( d \) from level \( k \) as before. For example, in Figure 6.24, reading a data object from block 12 in the level \( k \) cache would result in a cache miss because block 12 is not currently stored in the level \( k \) cache. Once it has been copied from level \( k + 1 \) to level \( k \), block 12 will remain there in expectation of later accesses.

Kinds of Cache Misses

It is sometimes helpful to distinguish between different kinds of cache misses. If the cache at level \( k \) is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a cold cache, and misses of this kind are called compulsory misses or cold misses. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been warmed up by repeated memory accesses.

Whenever there is a miss, the cache at level \( k \) must implement some placement policy that determines where to place the block it has retrieved from level \( k + 1 \). The most flexible placement policy is to allow any block from level \( k + 1 \) to be stored in any block at level \( k \). For caches high in the memory hierarchy (close to the CPU) that are implemented in hardware and where speed is at a premium, this policy is usually too expensive to implement because randomly placed blocks are expensive to locate.

Thus, hardware caches typically implement a more restricted placement policy that restricts a particular block at level \( k + 1 \) to a small subset (sometimes a singleton) of the blocks at level \( k \). For example, in Figure 6.24, we might decide that a block \( i \) at level \( k + 1 \) must be placed in block \( (i \mod 4) \) at level \( k \). For example, blocks 0, 4, 8, and 12 at level \( k + 1 \) would map to block 0 at level \( k \); blocks 1, 5, 9, and 13 would map to block 1; and so on. Notice that our example cache in Figure 6.24 uses this policy.

Restrictive placement policies of this kind lead to a type of miss known as a conflict miss, in which the cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing. For example, in Figure 6.24, if the program requests block 0, then block 8, then block 0, then block 8, and so on, each of the references to these two blocks would miss in the cache at level \( k \), even though this cache can hold a total of four blocks.
Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the *working set* of the phase. When the size of the working set exceeds the size of the cache, the cache will experience what are known as *capacity misses*. In other words, the cache is just too small to handle this particular working set.

**Cache Management**

As we have noted, the essence of the memory hierarchy is that the storage device at each level is a cache for the next lower level. At each level, some form of logic must *manage* the cache. By this we mean that something has to partition the cache storage into blocks, transfer blocks between different levels, decide when there are hits and misses, and then deal with them. The logic that manages the cache can be hardware, software, or a combination of the two.

For example, the compiler manages the register file, the highest level of the cache hierarchy. It decides when to issue loads when there are misses, and determines which register to store the data in. The caches at levels L1, L2, and L3 are managed entirely by hardware logic built into the caches. In a system with virtual memory, the DRAM main memory serves as a cache for data blocks stored on disk, and is managed by a combination of operating system software and address translation hardware on the CPU. For a machine with a distributed file system such as AFS, the local disk serves as a cache that is managed by the AFS client process running on the local machine. In most cases, caches operate automatically and do not require any specific or explicit actions from the program.

### 6.3.2 Summary of Memory Hierarchy Concepts

To summarize, memory hierarchies based on caching work because slower storage is cheaper than faster storage and because programs tend to exhibit locality:

- **Exploiting temporal locality.** Because of temporal locality, the same data objects are likely to be reused multiple times. Once a data object has been copied into the cache on the first miss, we can expect a number of subsequent hits on that object. Since the cache is faster than the storage at the next lower level, these subsequent hits can be served much faster than the original miss.

- **Exploiting spatial locality.** Blocks usually contain multiple data objects. Because of spatial locality, we can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

Caches are used everywhere in modern systems. As you can see from Figure 6.25, caches are used in CPU chips, operating systems, distributed file systems, and on the World Wide Web. They are built from and managed by various combinations of hardware and software. Note that there are a number of terms and acronyms in Figure 6.25 that we haven’t covered yet. We include them here to demonstrate how common caches are.


<table>
<thead>
<tr>
<th>Type</th>
<th>What cached</th>
<th>Where cached</th>
<th>Latency (cycles)</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU registers</td>
<td>4-byte or 8-byte word</td>
<td>On-chip CPU registers</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-chip TLB</td>
<td>0</td>
<td>Hardware MMU</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte block</td>
<td>On-chip L1 cache</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte block</td>
<td>On/off-chip L2 cache</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>L3 cache</td>
<td>64-byte block</td>
<td>On/off-chip L3 cache</td>
<td>30</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Controller firmware</td>
</tr>
<tr>
<td>Network cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>

Figure 6.25 The ubiquity of caching in modern computer systems. Acronyms: TLB: translation lookaside buffer, MMU: memory management unit, OS: operating system, AFS: Andrew File System, NFS: Network File System.

### 6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main DRAM memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert a small SRAM cache memory, called an **L1 cache** (Level 1 cache) between the CPU register file and main memory, as shown in Figure 6.26. The L1 cache can be accessed nearly as fast as the registers, typically in 2 to 4 clock cycles.

As the performance gap between the CPU and main memory continued to increase, system designers responded by inserting an additional larger cache, called an **L2 cache**, between the L1 cache and main memory, that can be accessed in about 10 clock cycles. Some modern systems include an additional even larger cache, called an **L3 cache**, which sits between the L2 cache and main memory.
in the memory hierarchy and can be accessed in 30 or 40 cycles. While there is considerable variety in the arrangements, the general principles are the same. For our discussion in the next section, we will assume a simple memory hierarchy with a single L1 cache between the CPU and main memory.

### 6.4.1 Generic Cache Memory Organization

Consider a computer system where each memory address has \( m \) bits that form \( M = 2^m \) unique addresses. As illustrated in Figure 6.27(a), a cache for such a machine is organized as an array of \( S = 2^s \) cache sets. Each set consists of \( E \) cache lines. Each line consists of a data block of \( B = 2^b \) bytes, a valid bit that indicates whether or not the line contains meaningful information, and \( t = m - (b + s) \) tag bits (a subset of the bits from the current block’s memory address) that uniquely identify the block stored in the cache line.

In general, a cache’s organization can be characterized by the tuple \((S, E, B, m)\). The size (or capacity) of a cache, \( C \), is stated in terms of the aggregate size of all the blocks. The tag bits and valid bit are not included. Thus, \( C = S \times E \times B \).

When the CPU is instructed by a load instruction to read a word from address \( A \) of main memory, it sends the address \( A \) to the cache. If the cache is holding a copy of the word at address \( A \), it sends the word immediately back to the CPU.

---

**Figure 6.27**

**General organization of cache** \((S, E, B, m)\).

(a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the \( m \) address bits into \( t \) tag bits, \( s \) set index bits, and \( b \) block offset bits.
Fundamental parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S = 2^x$</td>
<td>Number of sets</td>
</tr>
<tr>
<td>$E$</td>
<td>Number of lines per set</td>
</tr>
<tr>
<td>$B = 2^h$</td>
<td>Block size (bytes)</td>
</tr>
<tr>
<td>$m = \log_2(M)$</td>
<td>Number of physical (main memory) address bits</td>
</tr>
</tbody>
</table>

Derived quantities

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M = 2^m$</td>
<td>Maximum number of unique memory addresses</td>
</tr>
<tr>
<td>$s = \log_2(S)$</td>
<td>Number of set index bits</td>
</tr>
<tr>
<td>$b = \log_2(B)$</td>
<td>Number of block offset bits</td>
</tr>
<tr>
<td>$t = m - (s + b)$</td>
<td>Number of tag bits</td>
</tr>
<tr>
<td>$C = B \times E \times S$</td>
<td>Cache size (bytes) not including overhead such as the valid and tag bits</td>
</tr>
</tbody>
</table>

So how does the cache know whether it contains a copy of the word at address $A$? The cache is organized so that it can find the requested word by simply inspecting the bits of the address, similar to a hash table with an extremely simple hash function. Here is how it works:

The parameters $S$ and $B$ induce a partitioning of the $m$ address bits into the three fields shown in Figure 6.27(b). The $s$ set index bits in $A$ form an index into the array of $S$ sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the set index bits tell us which set the word must be stored in. Once we know which set the word must be contained in, the $t$ tag bits in $A$ tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address $A$. Once we have located the line identified by the tag in the set identified by the set index, then the $b$ block offset bits give us the offset of the word in the $B$-byte data block.

As you may have noticed, descriptions of caches use a lot of symbols. Figure 6.28 summarizes these symbols for your reference.

Practice Problem 6.10

The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets ($S$), tag bits ($t$), set index bits ($s$), and block offset bits ($b$).
### Section 6.4 Cache Memories

#### 6.4.2 Direct-Mapped Caches

Caches are grouped into different classes based on \( E \), the number of cache lines per set. A cache with exactly one line per set \( (E = 1) \) is known as a direct-mapped cache (see Figure 6.29). Direct-mapped caches are the simplest both to implement and to understand, so we will use them to illustrate some general concepts about how caches work.

Suppose we have a system with a CPU, a register file, an L1 cache, and a main memory. When the CPU executes an instruction that reads a memory word \( w \), it requests the word from the L1 cache. If the L1 cache has a cached copy of \( w \), then we have an L1 cache hit, and the cache quickly extracts \( w \) and returns it to the CPU. Otherwise, we have a cache miss, and the CPU must wait while the L1 cache requests a copy of the block containing \( w \) from the main memory. When the requested block finally arrives from memory, the L1 cache stores the block in one of its cache lines, extracts word \( w \) from the stored block, and returns it to the CPU. The process that a cache goes through of determining whether a request is a hit or a miss and then extracting the requested word consists of three steps: (1) set selection, (2) line matching, and (3) word extraction.

#### Set Selection in Direct-Mapped Caches

In this step, the cache extracts the \( s \) set index bits from the middle of the address for \( w \). These bits are interpreted as an unsigned integer that corresponds to a set number. In other words, if we think of the cache as a one-dimensional array of sets, then the set index bits form an index into this array. Figure 6.30 shows how set selection works for a direct-mapped cache. In this example, the set index bits 000012 are interpreted as an integer index that selects set 1.

#### Line Matching in Direct-Mapped Caches

Now that we have selected some set \( i \) in the previous step, the next step is to determine if a copy of the word \( w \) is stored in one of the cache lines contained in

---

<table>
<thead>
<tr>
<th>Cache</th>
<th>( m )</th>
<th>( C )</th>
<th>( B )</th>
<th>( E )</th>
<th>( S )</th>
<th>( t )</th>
<th>( s )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*Figure 6.29*  
Direct-mapped cache \( (E = 1) \). There is exactly one line per set.
set \( i \). In a direct-mapped cache, this is easy and fast because there is exactly one line per set. A copy of \( w \) is contained in the line if and only if the valid bit is set and the tag in the cache line matches the tag in the address of \( w \).

Figure 6.31 shows how line matching works in a direct-mapped cache. In this example, there is exactly one cache line in the selected set. The valid bit for this line is set, so we know that the bits in the tag and block are meaningful. Since the tag bits in the cache line match the tag bits in the address, we know that a copy of the word we want is indeed stored in the line. In other words, we have a cache hit. On the other hand, if either the valid bit were not set or the tags did not match, then we would have had a cache miss.

Word Selection in Direct-Mapped Caches

Once we have a hit, we know that \( w \) is somewhere in the block. This last step determines where the desired word starts in the block. As shown in Figure 6.31, the block offset bits provide us with the offset of the first byte in the desired word. Similar to our view of a cache as an array of lines, we can think of a block as an array of bytes, and the byte offset as an index into that array. In the example, the block offset bits of 1002 indicate that the copy of \( w \) starts at byte 4 in the block. (We are assuming that words are 4 bytes long.)

Line Replacement on Misses in Direct-Mapped Caches

If the cache misses, then it needs to retrieve the requested block from the next level in the memory hierarchy and store the new block in one of the cache lines of
the set indicated by the set index bits. In general, if the set is full of valid cache lines, then one of the existing lines must be evicted. For a direct-mapped cache, where each set contains exactly one line, the replacement policy is trivial: the current line is replaced by the newly fetched line.

**Putting It Together: A Direct-Mapped Cache in Action**

The mechanisms that a cache uses to select sets and identify lines are extremely simple. They have to be, because the hardware must perform them in a few nanoseconds. However, manipulating bits in this way can be confusing to us humans. A concrete example will help clarify the process. Suppose we have a direct-mapped cache described by

\[(S, E, B, m) = (4, 1, 2, 4)\]

In other words, the cache has four sets, one line per set, 2 bytes per block, and 4-bit addresses. We will also assume that each word is a single byte. Of course, these assumptions are totally unrealistic, but they will help us keep the example simple.

When you are first learning about caches, it can be very instructive to enumerate the entire address space and partition the bits, as we’ve done in Figure 6.32 for our 4-bit example. There are some interesting things to notice about this enumerated space:
The concatenation of the tag and index bits uniquely identifies each block in memory. For example, block 0 consists of addresses 0 and 1, block 1 consists of addresses 2 and 3, block 2 consists of addresses 4 and 5, and so on.

Since there are eight memory blocks but only four cache sets, multiple blocks map to the same cache set (i.e., they have the same set index). For example, blocks 0 and 4 both map to set 0, blocks 1 and 5 both map to set 1, and so on.

Blocks that map to the same cache set are uniquely identified by the tag. For example, block 0 has a tag bit of 0 while block 4 has a tag bit of 1, block 1 has a tag bit of 0 while block 5 has a tag bit of 1, and so on.

Let us simulate the cache in action as the CPU performs a sequence of reads. Remember that for this example, we are assuming that the CPU reads 1-byte words. While this kind of manual simulation is tedious and you may be tempted to skip it, in our experience students do not really understand how caches work until they work their way through a few of them.

Initially, the cache is empty (i.e., each valid bit is zero):

<table>
<thead>
<tr>
<th>Set</th>
<th>Valid</th>
<th>Tag</th>
<th>block[0]</th>
<th>block[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>m[0]</td>
<td>m[1]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each row in the table represents a cache line. The first column indicates the set that the line belongs to, but keep in mind that this is provided for convenience and is not really part of the cache. The next three columns represent the actual bits in each cache line. Now, let us see what happens when the CPU performs a sequence of reads:

1. **Read word at address 0.** Since the valid bit for set 0 is zero, this is a cache miss. The cache fetches block 0 from memory (or a lower-level cache) and stores the block in set 0. Then the cache returns m[0] (the contents of memory location 0) from block[0] of the newly fetched cache line.

<table>
<thead>
<tr>
<th>Set</th>
<th>Valid</th>
<th>Tag</th>
<th>block[0]</th>
<th>block[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>m[0]</td>
<td>m[1]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. **Read word at address 1.** This is a cache hit. The cache immediately returns m[1] from block[1] of the cache line. The state of the cache does not change.

3. **Read word at address 13.** Since the cache line in set 2 is not valid, this is a cache miss. The cache loads block 6 into set 2 and returns m[13] from block[1] of the new cache line.
4. **Read word at address 8.** This is a miss. The cache line in set 0 is indeed valid, but the tags do not match. The cache loads block 4 into set 0 (replacing the line that was there from the read of address 0) and returns m[8] from block[0] of the new cache line.

5. **Read word at address 0.** This is another miss, due to the unfortunate fact that we just replaced block 0 during the previous reference to address 8. This kind of miss, where we have plenty of room in the cache but keep alternating references to blocks that map to the same set, is an example of a conflict miss.

---

**Conflict Misses in Direct-Mapped Caches**

Conflict misses are common in real programs and can cause baffling performance problems. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of 2. For example, consider a function that computes the dot product of two vectors:

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0.0;
    int i;
    for (i = 0; i < 8; i++)
        sum += x[i] * y[i];
    return sum;
}
```
This function has good spatial locality with respect to \( x \) and \( y \), and so we might expect it to enjoy a good number of cache hits. Unfortunately, this is not always true.

Suppose that floats are 4 bytes, that \( x \) is loaded into the 32 bytes of contiguous memory starting at address 0, and that \( y \) starts immediately after \( x \) at address 32. For simplicity, suppose that a block is 16 bytes (big enough to hold four floats) and that the cache consists of two sets, for a total cache size of 32 bytes. We will assume that the variable \( \text{sum} \) is actually stored in a CPU register and thus does not require a memory reference. Given these assumptions, each \( x[i] \) and \( y[i] \) will map to the identical cache set:

<table>
<thead>
<tr>
<th>Element</th>
<th>Address</th>
<th>Set index</th>
<th>Element</th>
<th>Address</th>
<th>Set index</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x[0] )</td>
<td>0</td>
<td>0</td>
<td>( y[0] )</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>( x[1] )</td>
<td>4</td>
<td>0</td>
<td>( y[1] )</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>( x[2] )</td>
<td>8</td>
<td>0</td>
<td>( y[2] )</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>( x[3] )</td>
<td>12</td>
<td>0</td>
<td>( y[3] )</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>( x[4] )</td>
<td>16</td>
<td>1</td>
<td>( y[4] )</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>( x[5] )</td>
<td>20</td>
<td>1</td>
<td>( y[5] )</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>( x[6] )</td>
<td>24</td>
<td>1</td>
<td>( y[6] )</td>
<td>56</td>
<td>1</td>
</tr>
<tr>
<td>( x[7] )</td>
<td>28</td>
<td>1</td>
<td>( y[7] )</td>
<td>60</td>
<td>1</td>
</tr>
</tbody>
</table>

At run time, the first iteration of the loop references \( x[0] \), a miss that causes the block containing \( x[0]–x[3] \) to be loaded into set 0. The next reference is to \( y[0] \), another miss that causes the block containing \( y[0]–y[3] \) to be copied into set 0, overwriting the values of \( x \) that were copied in by the previous reference. During the next iteration, the reference to \( x[1] \) misses, which causes the \( x[0]–x[3] \) block to be loaded back into set 0, overwriting the \( y[0]–y[3] \) block. So now we have a conflict miss, and in fact each subsequent reference to \( x \) and \( y \) will result in a conflict miss as we thrash back and forth between blocks of \( x \) and \( y \). The term "thrashing" describes any situation where a cache is repeatedly loading and evicting the same sets of cache blocks.

The bottom line is that even though the program has good spatial locality and we have room in the cache to hold the blocks for both \( x[i] \) and \( y[i] \), each reference results in a conflict miss because the blocks map to the same cache set. It is not unusual for this kind of thrashing to result in a slowdown by a factor of 2 or 3. Also, be aware that even though our example is extremely simple, the problem is real for larger and more realistic direct-mapped caches.

Luckily, thrashing is easy for programmers to fix once they recognize what is going on. One easy solution is to put \( B \) bytes of padding at the end of each array. For example, instead of defining \( x \) to be \( \text{float} \ x[8] \), we define it to be \( \text{float} \ x[12] \). Assuming \( y \) starts immediately after \( x \) in memory, we have the following mapping of array elements to sets:
Section 6.4 Cache Memories

With the padding at the end of $x$, $x[i]$ and $y[i]$ now map to different sets, which eliminates the thrashing conflict misses.

**Practice Problem 6.11**

In the previous dotprod example, what fraction of the total references to $x$ and $y$ will be hits once we have padded array $x$?

**Practice Problem 6.12**

In general, if the high-order $s$ bits of an address are used as the set index, contiguous chunks of memory blocks are mapped to the same cache set.

A. How many blocks are in each of these contiguous array chunks?

B. Consider the following code that runs on a system with a cache of the form $(S, E, B, m) = (512, 1, 32, 32)$:

```c
int array[4096];

for (i = 0; i < 4096; i++)
    sum += array[i];
```

What is the maximum number of array blocks that are stored in the cache at any point in time?

**Aside**  Why index with the middle bits?

You may be wondering why caches use the middle bits for the set index instead of the high-order bits. There is a good reason why the middle bits are better. Figure 6.33 shows why. If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set. For example, in the figure, the first four blocks map to the first cache set, the second four blocks map to the second set, and so on. If a program has good spatial locality and scans the elements of an array sequentially, then the cache can only hold a block-sized chunk of the array at any point in time. This is an inefficient use of the cache. Contrast this with middle-bit indexing, where adjacent blocks always map to different cache lines. In this case, the cache can hold an entire $C$-sized chunk of the array, where $C$ is the cache size.
6.4.3 Set Associative Caches

The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$). A set associative cache relaxes this constraint so each set holds more than one cache line. A cache with $1 < E < C/B$ is often called an $E$-way set associative cache. We will discuss the special case, where $E = C/B$, in the next section. Figure 6.34 shows the organization of a two-way set associative cache.

**Set associative cache**

($1 < E < C/B$). In a set associative cache, each set contains more than one line. This particular example shows a two-way set associative cache.

**Figure 6.33** Why caches index with the middle bits.
Set Selection in Set Associative Caches

Set selection is identical to a direct-mapped cache, with the set index bits identifying the set. Figure 6.35 summarizes this principle.

Line Matching and Word Selection in Set Associative Caches

Line matching is more involved in a set associative cache than in a direct-mapped cache because it must check the tags and valid bits of multiple lines in order to determine if the requested word is in the set. A conventional memory is an array of values that takes an address as input and returns the value stored at that address. An associative memory, on the other hand, is an array of (key, value) pairs that takes as input the key and returns a value from one of the (key, value) pairs that matches the input key. Thus, we can think of each set in a set associative cache as a small associative memory where the keys are the concatenation of the tag and valid bits, and the values are the contents of a block.

Figure 6.36 shows the basic idea of line matching in an associative cache. An important idea here is that any line in the set can contain any of the memory blocks.

![Diagram of set selection in a set associative cache](image-url)

**Figure 6.35**  Set selection in a set associative cache.

**Figure 6.36**  Line matching and word selection in a set associative cache.
that map to that set. So the cache must search each line in the set, searching for a valid line whose tag matches the tag in the address. If the cache finds such a line, then we have a hit and the block offset selects a word from the block, as before.

**Line Replacement on Misses in Set Associative Caches**

If the word requested by the CPU is not stored in any of the lines in the set, then we have a cache miss, and the cache must fetch the block that contains the word from memory. However, once the cache has retrieved the block, which line should it replace? Of course, if there is an empty line, then it would be a good candidate. But if there are no empty lines in the set, then we must choose one of the nonempty lines and hope that the CPU does not reference the replaced line anytime soon.

It is very difficult for programmers to exploit knowledge of the cache replacement policy in their codes, so we will not go into much detail about it here. The simplest replacement policy is to choose the line to replace at random. Other more sophisticated policies draw on the principle of locality to try to minimize the probability that the replaced line will be referenced in the near future. For example, a *least-frequently-used* (LFU) policy will replace the line that has been referenced the fewest times over some past time window. A *least-recently-used* (LRU) policy will replace the line that was last accessed the furthest in the past. All of these policies require additional time and hardware. But as we move further down the memory hierarchy, away from the CPU, the cost of a miss becomes more expensive and it becomes more worthwhile to minimize misses with good replacement policies.

### 6.4.4 Fully Associative Caches

A **fully associative cache** consists of a single set (i.e., $E = C/B$) that contains all of the cache lines. Figure 6.37 shows the basic organization.

**Set Selection in Fully Associative Caches**

Set selection in a fully associative cache is trivial because there is only one set, summarized in Figure 6.38. Notice that there are no set index bits in the address, which is partitioned into only a tag and a block offset.

**Line Matching and Word Selection in Fully Associative Caches**

Line matching and word selection in a fully associative cache work the same as with a set associative cache, as we show in Figure 6.39. The difference is mainly a question of scale. Because the cache circuitry must search for many matching...
tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast. As a result, fully associative caches are only appropriate for small caches, such as the translation lookaside buffers (TLBs) in virtual memory systems that cache page table entries (Section 9.6.2).

**Practice Problem 6.13**

The problems that follow will help reinforce your understanding of how caches work. Assume the following:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative \((E = 2)\), with a 4-byte block size \((B = 4)\) and eight sets \((S = 8)\).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.
The following figure shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

**CO**  The cache block offset

**CI**  The cache set index

**CT**  The cache tag

A. Address format (one bit per box):

<table>
<thead>
<tr>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache block offset (CO)</td>
<td>0x</td>
</tr>
<tr>
<td>Cache set index (CI)</td>
<td>0x</td>
</tr>
<tr>
<td>Cache tag (CT)</td>
<td>0x</td>
</tr>
<tr>
<td>Cache hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Cache byte returned</td>
<td>0x</td>
</tr>
</tbody>
</table>
Practice Problem 6.15
Repeat Problem 6.14 for memory address $0x0DD5$.

A. Address format (one bit per box):

<table>
<thead>
<tr>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
</table>
| Cache block offset (CO)        | $0x\_
| Cache set index (CI)           | $0x\_
| Cache tag (CT)                 | $0x\_
| Cache hit? (Y/N)               |       |
| Cache byte returned            | $0x\_|

Practice Problem 6.16
Repeat Problem 6.14 for memory address $0x1FE4$.

A. Address format (one bit per box):

<table>
<thead>
<tr>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
</table>
| Cache block offset (CO)        | $0x\_
| Cache set index (CI)           | $0x\_
| Cache tag (CT)                 | $0x\_
| Cache hit? (Y/N)               |       |
| Cache byte returned            | $0x\_|

Practice Problem 6.17
For the cache in Problem 6.13, list all of the hex memory addresses that will hit in set 3.

6.4.5 Issues with Writes
As we have seen, the operation of a cache with respect to reads is straightforward. First, look for a copy of the desired word $w$ in the cache. If there is a hit, return
w immediately. If there is a miss, fetch the block that contains w from the next lower level of the memory hierarchy, store the block in some cache line (possibly evicting a valid line), and then return w.

The situation for writes is a little more complicated. Suppose we write a word w that is already cached (a write hit). After the cache updates its copy of w, what does it do about updating the copy of w in the next lower level of the hierarchy? The simplest approach, known as write-through, is to immediately write w’s cache block to the next lower level. While simple, write-through has the disadvantage of causing bus traffic with every write. Another approach, known as write-back, defers the update as long as possible by writing the updated block to the next lower level only when it is evicted from the cache by the replacement algorithm. Because of locality, write-back can significantly reduce the amount of bus traffic, but it has the disadvantage of additional complexity. The cache must maintain an additional dirty bit for each cache line that indicates whether or not the cache block has been modified.

Another issue is how to deal with write misses. One approach, known as write-allocate, loads the corresponding block from the next lower level into the cache and then updates the cache block. Write-allocate tries to exploit spatial locality of writes, but it has the disadvantage that every miss results in a block transfer from the next lower level to cache. The alternative, known as no-write-allocate, bypasses the cache and writes the word directly to the next lower level. Write-through caches are typically no-write-allocate. Write-back caches are typically write-allocate.

Optimizing caches for writes is a subtle and difficult issue, and we are only scratching the surface here. The details vary from system to system and are often proprietary and poorly documented. To the programmer trying to write reasonably cache-friendly programs, we suggest adopting a mental model that assumes write-back write-allocate caches. There are several reasons for this suggestion.

As a rule, caches at lower levels of the memory hierarchy are more likely to use write-back instead of write-through because of the larger transfer times. For example, virtual memory systems (which use main memory as a cache for the blocks stored on disk) use write-back exclusively. But as logic densities increase, the increased complexity of write-back is becoming less of an impediment and we are seeing write-back caches at all levels of modern systems. So this assumption matches current trends. Another reason for assuming a write-back write-allocate approach is that it is symmetric to the way reads are handled, in that write-back write-allocate tries to exploit locality. Thus, we can develop our programs at a high level to exhibit good spatial and temporal locality rather than trying to optimize for a particular memory system.

### 6.4.6 Anatomy of a Real Cache Hierarchy

So far, we have assumed that caches hold only program data. But in fact, caches can hold instructions as well as data. A cache that holds instructions only is called an i-cache. A cache that holds program data only is called a d-cache. A cache that holds both instructions and data is known as a unified cache. Modern processors
include separate i-caches and d-caches. There are a number of reasons for this. With two separate caches, the processor can read an instruction word and a data word at the same time. I-caches are typically read-only, and thus simpler. The two caches are often optimized to different access patterns and can have different block sizes, associativities, and capacities. Also, having separate caches ensures that data accesses do not create conflict misses with instruction accesses, and vice versa, at the cost of a potential increase in capacity misses.

Figure 6.40 shows the cache hierarchy for the Intel Core i7 processor. Each CPU chip has four cores. Each core has its own private L1 i-cache, L1 d-cache, and L2 unified cache. All of the cores share an on-chip L3 unified cache. An interesting feature of this hierarchy is that all of the SRAM cache memories are contained in the CPU chip.

Figure 6.41 summarizes the basic characteristics of the Core i7 caches.

<table>
<thead>
<tr>
<th>Cache type</th>
<th>Access time (cycles)</th>
<th>Cache size (C)</th>
<th>Assoc. (E)</th>
<th>Block size (B)</th>
<th>Sets (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 i-cache</td>
<td>4</td>
<td>32 KB</td>
<td>8</td>
<td>64 B</td>
<td>64</td>
</tr>
<tr>
<td>L1 d-cache</td>
<td>4</td>
<td>32 KB</td>
<td>8</td>
<td>64 B</td>
<td>64</td>
</tr>
<tr>
<td>L2 unified cache</td>
<td>11</td>
<td>256 KB</td>
<td>8</td>
<td>64 B</td>
<td>512</td>
</tr>
<tr>
<td>L3 unified cache</td>
<td>30–40</td>
<td>8 MB</td>
<td>16</td>
<td>64 B</td>
<td>8192</td>
</tr>
</tbody>
</table>

Figure 6.41 Characteristics of the Intel Core i7 cache hierarchy.
6.4.7 Performance Impact of Cache Parameters

Cache performance is evaluated with a number of metrics:

- Miss rate. The fraction of memory references during the execution of a program, or a part of a program, that miss. It is computed as \#misses/\#references.
- Hit rate. The fraction of memory references that hit. It is computed as \(1 - \text{miss rate}\).
- Hit time. The time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is on the order of several clock cycles for L1 caches.
- Miss penalty. Any additional time required because of a miss. The penalty for L1 misses served from L2 is on the order of 10 cycles; from L3, 40 cycles; and from main memory, 100 cycles.

Optimizing the cost and performance trade-offs of cache memories is a subtle exercise that requires extensive simulation on realistic benchmark codes and thus is beyond our scope. However, it is possible to identify some of the qualitative trade-offs.

Impact of Cache Size

On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make large memories run faster. As a result, larger caches tend to increase the hit time. This is especially important for on-chip L1 caches that must have a short hit time.

Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems usually compromise with cache blocks that contain 32 to 64 bytes.

Impact of Associativity

The issue here is the impact of the choice of the parameter \(E\), the number of cache lines per set. The advantage of higher associativity (i.e., larger values of \(E\)) is that it decreases the vulnerability of the cache to thrashing due to conflict misses. However, higher associativity comes at a significant cost. Higher associativity is expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. Higher associativity can increase hit time, because of the increased complexity, and it can also increase the miss penalty because of the increased complexity of choosing a victim line.
The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for smaller associativity for L1 caches (where the miss penalty is only a few cycles) and a higher degree of associativity for the lower levels, where the miss penalty is higher. For example, in Intel Core i7 systems, the L1 and L2 caches are 8-way associative, and the L3 cache is 16-way.

Impact of Write Strategy

Write-through caches are simpler to implement and can use a write buffer that works independently of the cache to update memory. Furthermore, read misses are less expensive because they do not trigger a memory write. On the other hand, write-back caches result in fewer transfers, which allows more bandwidth to memory for I/O devices that perform DMA. Further, reducing the number of transfers becomes increasingly important as we move down the hierarchy and the transfer times increase. In general, caches further down the hierarchy are more likely to use write-back than write-through.

Aside  Cache lines, sets, and blocks: What’s the difference?

It is easy to confuse the distinction between cache lines, sets, and blocks. Let’s review these ideas and make sure they are clear:

- A block is a fixed-sized packet of information that moves back and forth between a cache and main memory (or a lower-level cache).
- A line is a container in a cache that stores a block, as well as other information such as the valid bit and the tag bits.
- A set is a collection of one or more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.

In direct-mapped caches, sets and lines are indeed equivalent. However, in associative caches, sets and lines are very different things and the terms cannot be used interchangeably.

Since a line always stores a single block, the terms “line” and “block” are often used interchangeably. For example, systems professionals usually refer to the “line size” of a cache, when what they really mean is the block size. This usage is very common, and shouldn’t cause any confusion, so long as you understand the distinction between blocks and lines.

6.5  Writing Cache-friendly Code

In Section 6.2, we introduced the idea of locality and talked in qualitative terms about what constitutes good locality. Now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to
write code that is \textit{cache friendly}, in the sense that it has good locality. Here is the basic approach we use to try to ensure that our code is cache friendly.

1. \textit{Make the common case go fast.} Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core functions and ignore the rest.

2. \textit{Minimize the number of cache misses in each inner loop.}\ All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

To see how this works in practice, consider the \texttt{sumvec} function from Section 6.2:

\begin{verbatim}
1 int sumvec(int v[N])
2 {
3   int i, sum = 0;
4
5   for (i = 0; i < N; i++)
6     sum += v[i];
7   return sum;
8 }
\end{verbatim}

Is this function cache friendly? First, notice that there is good temporal locality in the loop body with respect to the local variables \texttt{i} and \texttt{sum}. In fact, because these are local variables, any reasonable optimizing compiler will cache them in the register file, the highest level of the memory hierarchy. Now consider the stride-1 references to vector \texttt{v}. In general, if a cache has a block size of \(B\) bytes, then a stride-\(k\) reference pattern (where \(k\) is expressed in words) results in an average of \(\min\left(1, \frac{\text{wordsize} \times k}{B}\right)\) misses per loop iteration. This is minimized for \(k = 1\), so the stride-1 references to \texttt{v} are indeed cache friendly. For example, suppose that \texttt{v} is block aligned, words are 4 bytes, cache blocks are 4 words, and the cache is initially empty (a cold cache). Then, regardless of the cache organization, the references to \texttt{v} will result in the following pattern of hits and misses:

\begin{center}
\begin{tabular}{cccccccc}
\texttt{v[i]} & \texttt{i = 0} & \texttt{i = 1} & \texttt{i = 2} & \texttt{i = 3} & \texttt{i = 4} & \texttt{i = 5} & \texttt{i = 6} & \texttt{i = 7} \\
\hline
\texttt{[m]} & \texttt{[m]} & \texttt{[m]} & \texttt{[m]} & \texttt{[m]} & \texttt{[m]} & \texttt{[m]} & \texttt{[m]} & \texttt{[m]} \\
\end{tabular}
\end{center}

In this example, the reference to \texttt{v[0]} misses and the corresponding block, which contains \texttt{v[0]}–\texttt{v[3]}, is loaded into the cache from memory. Thus, the next three references are all hits. The reference to \texttt{v[4]} causes another miss as a new block is loaded into the cache, the next three references are hits, and so on. In general, three out of four references will hit, which is the best we can do in this case with a cold cache.

To summarize, our simple \texttt{sumvec} example illustrates two important points about writing cache-friendly code:

- Repeated references to local variables are good because the compiler can cache them in the register file (temporal locality).
• Stride-1 reference patterns are good because caches at all levels of the memory hierarchy store data as contiguous blocks (spatial locality).

Spatial locality is especially important in programs that operate on multi-dimensional arrays. For example, consider the `sumarrayrows` function from Section 6.2, which sums the elements of a two-dimensional array in row-major order:

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Since C stores arrays in row-major order, the inner loop of this function has the same desirable stride-1 access pattern as `sumvec`. For example, suppose we make the same assumptions about the cache as for `sumvec`. Then the references to the array `a` will result in the following pattern of hits and misses:

<table>
<thead>
<tr>
<th>a[i][j]</th>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 0</td>
<td>1[m]</td>
<td>9[m]</td>
<td>17[m]</td>
<td>25[m]</td>
</tr>
<tr>
<td>j = 1</td>
<td>2[h]</td>
<td>10[h]</td>
<td>18[h]</td>
<td>26[h]</td>
</tr>
<tr>
<td>j = 2</td>
<td>3[h]</td>
<td>11[h]</td>
<td>19[h]</td>
<td>27[h]</td>
</tr>
<tr>
<td>j = 3</td>
<td>4[h]</td>
<td>12[h]</td>
<td>20[h]</td>
<td>28[h]</td>
</tr>
<tr>
<td>j = 4</td>
<td>5[m]</td>
<td>13[m]</td>
<td>21[m]</td>
<td>29[m]</td>
</tr>
<tr>
<td>j = 5</td>
<td>6[h]</td>
<td>14[h]</td>
<td>22[h]</td>
<td>30[h]</td>
</tr>
<tr>
<td>j = 6</td>
<td>7[h]</td>
<td>15[h]</td>
<td>23[h]</td>
<td>31[h]</td>
</tr>
<tr>
<td>j = 7</td>
<td>8[h]</td>
<td>16[h]</td>
<td>24[h]</td>
<td>32[h]</td>
</tr>
</tbody>
</table>

But consider what happens if we make the seemingly innocuous change of permuting the loops:

```c
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

In this case, we are scanning the array column by column instead of row by row. If we are lucky and the entire array fits in the cache, then we will enjoy the same miss rate of 1/4. However, if the array is larger than the cache (the more likely case), then each and every access of `a[i][j]` will miss!
Higher miss rates can have a significant impact on running time. For example, on our desktop machine, sumarrayrows runs twice as fast as sumarraycols. To summarize, programmers should be aware of locality in their programs and try to write programs that exploit it.

**Practice Problem 6.18**

Transposing the rows and columns of a matrix is an important problem in signal processing and scientific computing applications. It is also interesting from a locality point of view because its reference pattern is both row-wise and column-wise. For example, consider the following transpose routine:

```c
typedef int array[2][2];

void transpose1(array dst, array src)
{
    int i, j;

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {
            dst[j][i] = src[i][j];
        }
    }
}
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with a block size of 8 bytes.
- The cache has a total size of 16 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. For each row and col, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.
Practice Problem 6.19

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1024-byte direct-mapped data cache with 16-byte blocks ($B = 16$). You are given the following definitions:

```c
struct algae_position {
    int x;
    int y;
};
struct algae_position grid[16][16];
int total_x = 0, total_y = 0;
int i, j;
```

You should also assume the following:

- `sizeof(int) == 4`
- `grid` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `grid`. Variables `i`, `j`, `total_x`, and `total_y` are stored in registers.

Determine the cache performance for the following code:

```c
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        total_x += grid[i][j].x;
    }
}
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        total_y += grid[i][j].y;
    }
}
```
A. What is the total number of reads?
B. What is the total number of reads that miss in the cache?
C. What is the miss rate?

Practice Problem 6.20
Given the assumptions of Problem 6.19, determine the cache performance of the following code:

```c
1 for (i = 0; i < 16; i++){
2     for (j = 0; j < 16; j++) {
3         total_x += grid[j][i].x;
4         total_y += grid[j][i].y;
5     }
6 }
```

A. What is the total number of reads?
B. What is the total number of reads that miss in the cache?
C. What is the miss rate?
D. What would the miss rate be if the cache were twice as big?

Practice Problem 6.21
Given the assumptions of Problem 6.19, determine the cache performance of the following code:

```c
1 for (i = 0; i < 16; i++){
2     for (j = 0; j < 16; j++) {
3         total_x += grid[i][j].x;
4         total_y += grid[i][j].y;
5     }
6 }
```

A. What is the total number of reads?
B. What is the total number of reads that miss in the cache?
C. What is the miss rate?
D. What would the miss rate be if the cache were twice as big?

6.6 Putting It Together: The Impact of Caches on Program Performance

This section wraps up our discussion of the memory hierarchy by studying the impact that caches have on the performance of programs running on real machines.
6.6.1 The Memory Mountain

The rate that a program reads data from the memory system is called the read throughput, or sometimes the read bandwidth. If a program reads \( n \) bytes over a period of \( s \) seconds, then the read throughput over that period is \( n/s \), typically expressed in units of megabytes per second (MB/s).

If we were to write a program that issued a sequence of read requests from a tight program loop, then the measured read throughput would give us some insight into the performance of the memory system for that particular sequence of reads. Figure 6.42 shows a pair of functions that measure the read throughput for a particular read sequence.

The test function generates the read sequence by scanning the first \( \text{elems} \) elements of an array with a stride of \( \text{stride} \). The run function is a wrapper that calls the test function and returns the measured read throughput. The call to the test function in line 29 warms the cache. The \text{fcyc2} function in line 30 calls the test function with arguments \( \text{elems} \) and estimates the running time of the test function in CPU cycles. Notice that the size argument to the run function is in units of bytes, while the corresponding \( \text{elems} \) argument to the test function is in units of array elements. Also, notice that line 31 computes MB/s as \( 10^6 \) bytes/s, as opposed to \( 2^{20} \) bytes/s.

The size and stride arguments to the run function allow us to control the degree of temporal and spatial locality in the resulting read sequence. Smaller values of size result in a smaller working set size, and thus better temporal locality. Smaller values of stride result in better spatial locality. If we call the run function repeatedly with different values of size and stride, then we can recover a fascinating two-dimensional function of read throughput versus temporal and spatial locality. This function is called a memory mountain.

Every computer has a unique memory mountain that characterizes the capabilities of its memory system. For example, Figure 6.43 shows the memory mountain for an Intel Core i7 system. In this example, the size varies from 2 KB to 64 MB, and the stride varies from 1 to 64 elements, where each element is an 8-byte double.

The geography of the Core i7 mountain reveals a rich structure. Perpendicular to the size axis are four ridges that correspond to the regions of temporal locality where the working set fits entirely in the L1 cache, the L2 cache, the L3 cache, and main memory, respectively. Notice that there is an order of magnitude difference between the highest peak of the L1 ridge, where the CPU reads at a rate of over 6 GB/s, and the lowest point of the main memory ridge, where the CPU reads at a rate of 600 MB/s.

There is a feature of the L1 ridge that should be pointed out. For very large strides, notice how the read throughput drops as the working set size approaches 2 KB (falling off the back side of the ridge). Since the L1 cache holds the entire working set, this feature does not reflect the true L1 cache performance. It is an artifact of overheads of calling the test function and setting up to execute the loop. For large strides in small working set sizes, these overheads are not amortized, as they are with the larger sizes.
double data[MAXELEMS]; /* The global array we'll be traversing */

/*
test - Iterate over first "elems" elements of array "data"
with stride of "stride".
*/
void test(int elems, int stride) /* The test function */
{
    int i;
    double result = 0.0;
    volatile double sink;
    for (i = 0; i < elems; i += stride) {
        result += data[i];
    }
    sink = result; /* So compiler doesn't optimize away the loop */
}

/*
run - Run test(elems, stride) and return read throughput (MB/s).
"size" is in bytes, "stride" is in array elements, and
Mhz is CPU clock frequency in Mhz.
*/
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(double);
    test(elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}

**Figure 6.42** Functions that measure and compute read throughput. We can generate a memory mountain for a particular computer by calling the run function with different values of size (which corresponds to temporal locality) and stride (which corresponds to spatial locality).

On each of the L2, L3, and main memory ridges, there is a slope of spatial locality that falls downhill as the stride increases, and spatial locality decreases. Notice that even when the working set is too large to fit in any of the caches, the highest point on the main memory ridge is a factor of 7 higher than its lowest point. So even when a program has poor temporal locality, spatial locality can still come to the rescue and make a significant difference.
There is a particularly interesting flat ridge line that extends perpendicular to the stride axis for strides of 1 and 2, where the read throughput is a relatively constant 4.5 GB/s. This is apparently due to a hardware *prefetching* mechanism in the Core i7 memory system that automatically identifies memory referencing patterns and attempts to fetch those blocks into cache before they are accessed. While the details of the particular prefetching algorithm are not documented, it is clear from the memory mountain that the algorithm works best for small strides—yet another reason to favor sequential accesses in your code.

If we take a slice through the mountain, holding the stride constant as in Figure 6.44, we can see the impact of cache size and temporal locality on performance. For sizes up to 32 KB, the working set fits entirely in the L1 d-cache, and thus reads are served from L1 at the peak throughput of about 6 GB/s. For sizes up to 256 KB, the working set fits entirely in the unified L2 cache, and for sizes up to 8M, the working set fits entirely in the unified L3 cache. Larger working set sizes are served primarily from main memory.

The dips in read throughputs at the leftmost edges of the L1, L2, and L3 cache regions—where the working set sizes of 32 KB, 256 KB, and 8 MB are equal to their respective cache sizes—are interesting. It is not entirely clear why these dips occur. The only way to be sure is to perform a detailed cache simulation, but it
is likely that the drops are caused by other data and code blocks that make it impossible to fit the entire array in the respective cache.

Slicing through the memory mountain in the opposite direction, holding the working set size constant, gives us some insight into the impact of spatial locality on the read throughput. For example, Figure 6.45 shows the slice for a fixed working set size of 4 MB. This slice cuts along the L3 ridge in Figure 6.43, where the working set fits entirely in the L3 cache, but is too large for the L2 cache.

Notice how the read throughput decreases steadily as the stride increases from one to eight doublewords. In this region of the mountain, a read miss in L2 causes a block to be transferred from L3 to L2. This is followed by some number of hits on the block in L2, depending on the stride. As the stride increases, the ratio of L2 misses to L2 hits increases. Since misses are served more slowly than hits, the read throughput decreases. Once the stride reaches eight doublewords, which on this system equals the block size of 64 bytes, every read request misses in L2 and must be served from L3. Thus, the read throughput for strides of at least eight doublewords is a constant rate determined by the rate that cache blocks can be transferred from L3 into L2.

To summarize our discussion of the memory mountain, the performance of the memory system is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations can vary by over an order of magnitude. Wise programmers try to structure their programs so that they run in the peaks instead of the valleys. The aim is to exploit temporal locality so that
Figure 6.45 A slope of spatial locality. The graph shows a slice through Figure 6.43 with size=4 MB.

heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.

Practice Problem 6.22
Use the memory mountain in Figure 6.43 to estimate the time, in CPU cycles, to read an 8-byte word from the L1 d-cache.

6.6.2 Rearranging Loops to Increase Spatial Locality

Consider the problem of multiplying a pair of $n \times n$ matrices: $C = AB$. For example, if $n = 2$, then

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

where

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

A matrix multiply function is usually implemented using three nested loops, which are identified by their indexes $i$, $j$, and $k$. If we permute the loops and make some other minor code changes, we can create the six functionally equivalent versions
of matrix multiply shown in Figure 6.46. Each version is uniquely identified by the ordering of its loops.

At a high level, the six versions are quite similar. If addition is associative, then each version computes an identical result. As we learned in Chapter 2, floating-point addition is commutative, but in general not associative. In practice, if the matrices do not mix extremely large values with extremely small ones, as often is true when the matrices store physical properties, then the assumption of associativity is reasonable.

2. As we learned in Chapter 2, floating-point addition is commutative, but in general not associative.
Matrix multiply version (class) | Loads per iter. | Stores per iter. | A misses per iter. | B misses per iter. | C misses per iter. | Total misses per iter.
---|---|---|---|---|---|---
ijk & jik (AB) | 2 | 0 | 0.25 | 1.00 | 0.00 | 1.25
jki & kji (AC) | 2 | 1 | 1.00 | 0.00 | 1.00 | 2.00
kij & ikj (BC) | 2 | 1 | 0.00 | 0.25 | 0.25 | 0.50

Figure 6.47 Analysis of matrix multiply inner loops. The six versions partition into three equivalence classes, denoted by the pair of arrays that are accessed in the inner loop.

operations and an identical number of adds and multiplies. Each of the $n^2$ elements of $A$ and $B$ is read $n$ times. Each of the $n^2$ elements of $C$ is computed by summing $n$ values. However, if we analyze the behavior of the innermost loop iterations, we find that there are differences in the number of accesses and the locality. For the purposes of this analysis, we make the following assumptions:

- Each array is an $n \times n$ array of double, with `sizeof(double) == 8`.
- There is a single cache with a 32-byte block size ($B = 32$).
- The array size $n$ is so large that a single matrix row does not fit in the L1 cache.
- The compiler stores local variables in registers, and thus references to local variables inside loops do not require any load or store instructions.

Figure 6.47 summarizes the results of our inner loop analysis. Notice that the six versions pair up into three equivalence classes, which we denote by the pair of matrices that are accessed in the inner loop. For example, versions $ijk$ and $jik$ are members of Class $AB$ because they reference arrays $A$ and $B$ (but not $C$) in their innermost loop. For each class, we have counted the number of loads (reads) and stores (writes) in each inner loop iteration, the number of references to $A$, $B$, and $C$ that will miss in the cache in each loop iteration, and the total number of cache misses per iteration.

The inner loops of the Class $AB$ routines (Figure 6.46(a) and (b)) scan a row of array $A$ with a stride of 1. Since each cache block holds four doublewords, the miss rate for $A$ is 0.25 misses per iteration. On the other hand, the inner loop scans a column of $B$ with a stride of $n$. Since $n$ is large, each access of array $B$ results in a miss, for a total of 1.25 misses per iteration.

The inner loops in the Class $AC$ routines (Figure 6.46(c) and (d)) have some problems. Each iteration performs two loads and a store (as opposed to the Class $AB$ routines, which perform two loads and no stores). Second, the inner loop scans the columns of $A$ and $C$ with a stride of $n$. The result is a miss on each load, for a total of two misses per iteration. Notice that interchanging the loops has decreased the amount of spatial locality compared to the Class $AB$ routines.

The $BC$ routines (Figure 6.46(e) and (f)) present an interesting trade-off: With two loads and a store, they require one more memory operation than the $AB$ routines. On the other hand, since the inner loop scans both $B$ and $C$ row-wise
Figure 6.48
Core i7 matrix multiply performance. Legend: jki and kji: Class AC; ijk and jik: Class AB; kij and ikj: Class BC.

with a stride-1 access pattern, the miss rate on each array is only 0.25 misses per iteration, for a total of 0.50 misses per iteration.

Figure 6.48 summarizes the performance of different versions of matrix multiply on a Core i7 system. The graph plots the measured number of CPU cycles per inner loop iteration as a function of array size (n).

There are a number of interesting points to notice about this graph:

- For large values of \( n \), the fastest version runs almost 20 times faster than the slowest version, even though each performs the same number of floating-point arithmetic operations.
- Pairs of versions with the same number of memory references and misses per iteration have almost identical measured performance.
- The two versions with the worst memory behavior, in terms of the number of accesses and misses per iteration, run significantly slower than the other four versions, which have fewer misses or fewer accesses, or both.
- Miss rate, in this case, is a better predictor of performance than the total number of memory accesses. For example, the Class BC routines, with 0.5 misses per iteration, perform much better than the Class AB routines, with 1.25 misses per iteration, even though the Class BC routines perform more memory references in the inner loop (two loads and one store) than the Class AB routines (two loads).
- For large values of \( n \), the performance of the fastest pair of versions (kij and ikj) is constant. Even though the array is much larger than any of the SRAM cache memories, the prefetching hardware is smart enough to recognize the stride-1 access pattern, and fast enough to keep up with memory accesses in the tight inner loop. This is a stunning accomplishment by the Intel engi-
neers who designed this memory system, providing even more incentive for programmers to develop programs with good spatial locality.

**Web Aside MEM:BLOCKING** Using blocking to increase temporal locality

There is an interesting technique called blocking that can improve the temporal locality of inner loops. The general idea of blocking is to organize the data structures in a program into large chunks called blocks. (In this context, “block” refers to an application-level chunk of data, not to a cache block.) The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on.

Unlike the simple loop transformations for improving spatial locality, blocking makes the code harder to read and understand. For this reason, it is best suited for optimizing compilers or frequently executed library routines. Still, the technique is interesting to study and understand because it is a general concept that can produce big performance gains on some systems.

### 6.6.3 Exploiting Locality in Your Programs

As we have seen, the memory system is organized as a hierarchy of storage devices, with smaller, faster devices toward the top and larger, slower devices toward the bottom. Because of this hierarchy, the effective rate that a program can access memory locations is not characterized by a single number. Rather, it is a wildly varying function of program locality (what we have dubbed the memory mountain) that can vary by orders of magnitude. Programs with good locality access most of their data from fast cache memories. Programs with poor locality access most of their data from the relatively slow DRAM main memory.

Programmers who understand the nature of the memory hierarchy can exploit this understanding to write more efficient programs, regardless of the specific memory system organization. In particular, we recommend the following techniques:

- Focus your attention on the inner loops, where the bulk of the computations and memory accesses occur.
- Try to maximize the spatial locality in your programs by reading data objects sequentially, with stride 1, in the order they are stored in memory.
- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.

### 6.7 Summary

The basic storage technologies are random-access memories (RAMs), nonvolatile memories (ROMs), and disks. RAM comes in two basic forms. Static RAM (SRAM) is faster and more expensive, and is used for cache memories both on and off the CPU chip. Dynamic RAM (DRAM) is slower and less expensive, and is used for the main memory and graphics frame buffers. Nonvolatile memories, also called read-only memories (ROMs), retain their information even if the supply voltage is turned off, and they are used to store firmware. Rotating disks are
mechanical nonvolatile storage devices that hold enormous amounts of data at a low cost per bit, but with much longer access times than DRAM. Solid state disks (SSDs) based on nonvolatile flash memory are becoming increasingly attractive alternatives to rotating disks for some applications.

In general, faster storage technologies are more expensive per bit and have smaller capacities. The price and performance properties of these technologies are changing at dramatically different rates. In particular, DRAM and disk access times are much larger than CPU cycle times. Systems bridge these gaps by organizing memory as a hierarchy of storage devices, with smaller, faster devices at the top and larger, slower devices at the bottom. Because well-written programs have good locality, most data are served from the higher levels, and the effect is a memory system that runs at the rate of the higher levels, but at the cost and capacity of the lower levels.

Programmers can dramatically improve the running times of their programs by writing programs with good spatial and temporal locality. Exploiting SRAM-based cache memories is especially important. Programs that fetch data primarily from cache memories can run much faster than programs that fetch data primarily from memory.

Bibliographic Notes

Memory and disk technologies change rapidly. In our experience, the best sources of technical information are the Web pages maintained by the manufacturers. Companies such as Micron, Toshiba, and Samsung provide a wealth of current technical information on memory devices. The pages for Seagate, Maxtor, and Western Digital provide similarly useful information about disks.

Textbooks on circuit and logic design provide detailed information about memory technology [56, 85]. IEEE Spectrum published a series of survey articles on DRAM [53]. The International Symposium on Computer Architecture (ISCA) is a common forum for characterizations of DRAM memory performance [34, 35].

Wilkes wrote the first paper on cache memories [116]. Smith wrote a classic survey [101]. Przybylski wrote an authoritative book on cache design [82]. Hennessy and Patterson provide a comprehensive discussion of cache design issues [49].

Stricker introduced the idea of the memory mountain as a comprehensive characterization of the memory system in [111], and suggested the term “memory mountain” informally in later presentations of the work. Compiler researchers work to increase locality by automatically performing the kinds of manual code transformations we discussed in Section 6.6 [22, 38, 63, 68, 75, 83, 118]. Carter and colleagues have proposed a cache-aware memory controller [18]. Seward developed an open-source cache profiler, called cacheprof, that characterizes the miss behavior of C programs on an arbitrary simulated cache (www.cacheprof.org). Other researchers have developed cache oblivious algorithms that are designed to run well without any explicit knowledge of the structure of the underlying cache memory [36, 42, 43].
There is a large body of literature on building and using disk storage. Many storage researchers look for ways to aggregate individual disks into larger, more robust, and more secure storage pools [20, 44, 45, 79, 119]. Others look for ways to use caches and locality to improve the performance of disk accesses [12, 21]. Systems such as Exokernel provide increased user-level control of disk and memory resources [55]. Systems such as the Andrew File System [74] and Coda [91] extend the memory hierarchy across computer networks and mobile notebook computers. Schindler and Ganger developed an interesting tool that automatically characterizes the geometry and performance of SCSI disk drives [92]. Researchers are investigating techniques for building and using Flash-based SSDs [8, 77].

**Homework Problems**

6.23◆◆
Suppose you are asked to design a rotating disk where the number of bits per track is constant. You know that the number of bits per track is determined by the circumference of the innermost track, which you can assume is also the circumference of the hole. Thus, if you make the hole in the center of the disk larger, the number of bits per track increases, but the total number of tracks decreases. If you let \( r \) denote the radius of the platter, and \( x \cdot r \) the radius of the hole, what value of \( x \) maximizes the capacity of the disk?

6.24◆
Estimate the average time (in ms) to access a sector on the following disk:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotational rate</td>
<td>15,000 RPM</td>
</tr>
<tr>
<td>( T_{avg\ seek} )</td>
<td>4 ms</td>
</tr>
<tr>
<td>Average # sectors/track</td>
<td>800</td>
</tr>
</tbody>
</table>

6.25◆◆
Suppose that a 2 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotational rate</td>
<td>15,000 RPM</td>
</tr>
<tr>
<td>( T_{avg\ seek} )</td>
<td>4 ms</td>
</tr>
<tr>
<td>Average # sectors/track</td>
<td>1000</td>
</tr>
<tr>
<td>Surfaces</td>
<td>8</td>
</tr>
<tr>
<td>Sector size</td>
<td>512 bytes</td>
</tr>
</tbody>
</table>

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is \( T_{avg\ seek} + T_{avg\ rotation} \).
6.26 ◆
The following table gives the parameters for a number of different caches. For each cache, fill in the missing fields in the table. Recall that \( m \) is the number of physical address bits, \( C \) is the cache size (number of data bytes), \( B \) is the block size in bytes, \( E \) is the associativity, \( S \) is the number of cache sets, \( t \) is the number of tag bits, \( s \) is the number of set index bits, and \( b \) is the number of block offset bits.

<table>
<thead>
<tr>
<th>Cache</th>
<th>( m )</th>
<th>( C )</th>
<th>( B )</th>
<th>( E )</th>
<th>( S )</th>
<th>( t )</th>
<th>( s )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.27 ◆
The following table gives the parameters for a number of different caches. Your task is to fill in the missing fields in the table. Recall that \( m \) is the number of physical address bits, \( C \) is the cache size (number of data bytes), \( B \) is the block size in bytes, \( E \) is the associativity, \( S \) is the number of cache sets, \( t \) is the number of tag bits, \( s \) is the number of set index bits, and \( b \) is the number of block offset bits.

<table>
<thead>
<tr>
<th>Cache</th>
<th>( m )</th>
<th>( C )</th>
<th>( B )</th>
<th>( E )</th>
<th>( S )</th>
<th>( t )</th>
<th>( s )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>32</td>
<td></td>
<td>8</td>
<td>1</td>
<td></td>
<td>21</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>32</td>
<td>2048</td>
<td></td>
<td></td>
<td></td>
<td>128</td>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td>3.</td>
<td>32</td>
<td>1024</td>
<td>2</td>
<td>8</td>
<td>64</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4.</td>
<td>32</td>
<td>1024</td>
<td></td>
<td>2</td>
<td>16</td>
<td>23</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

6.28 ◆
This problem concerns the cache in Problem 6.13.

A. List all of the hex memory addresses that will hit in set 1.
B. List all of the hex memory addresses that will hit in set 6.

6.29 ◆◆
This problem concerns the cache in Problem 6.13.

A. List all of the hex memory addresses that will hit in set 2.
B. List all of the hex memory addresses that will hit in set 4.
C. List all of the hex memory addresses that will hit in set 5.
D. List all of the hex memory addresses that will hit in set 7.

6.30  
Suppose we have a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 12 bits wide.
- The cache is two-way set associative \((E = 2)\), with a 4-byte block size \((B = 4)\) and four sets \((S = 4)\).

The contents of the cache are as follows, with all addresses, tags, and values given in hexadecimal notation:

<table>
<thead>
<tr>
<th>Set index</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>1</td>
<td>FE</td>
<td>97</td>
<td>CC</td>
<td>D0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>1</td>
<td>48</td>
<td>49</td>
<td>4A</td>
<td>4B</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>FF</td>
<td>1</td>
<td>9A</td>
<td>C0</td>
<td>03</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

A. The following diagram shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

- \(CO\) The cache block offset
- \(CI\) The cache set index
- \(CT\) The cache tag

<table>
<thead>
<tr>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

B. For each of the following memory accesses indicate if it will be a cache hit or miss when carried out in sequence as listed. Also give the value of a read if it can be inferred from the information in the cache.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address</th>
<th>Hit?</th>
<th>Read value (or unknown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>0x834</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>0x836</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>0xFFD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.31 Suppose we have a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is four-way set associative \((E = 4)\), with a 4-byte block size \((B = 4)\) and eight sets \((S = 8)\).

Consider the following cache state. All addresses, tags, and values are given in hexadecimal format. The Index column contains the set index for each set of four lines. The Tag columns contain the tag value for each line. The V columns contain the valid bit for each line. The Bytes 0–3 columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag V</th>
<th>Bytes 0–3</th>
<th>Tag V</th>
<th>Bytes 0–3</th>
<th>Tag V</th>
<th>Bytes 0–3</th>
<th>Tag V</th>
<th>Bytes 0–3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0 1</td>
<td>ED 32 0A A2</td>
<td>8A 1</td>
<td>BF 80 1D FC</td>
<td>14 1</td>
<td>EF 09 86 2A</td>
<td>BC 0</td>
<td>25 44 6F 1A</td>
</tr>
<tr>
<td>1</td>
<td>BC 0</td>
<td>03 3E CD 38</td>
<td>A0 0</td>
<td>16 7B ED 5A</td>
<td>BC 1</td>
<td>8E 4C DF 18</td>
<td>E4 1</td>
<td>FB B7 12 02</td>
</tr>
<tr>
<td>2</td>
<td>BC 1</td>
<td>54 9E 1E FA</td>
<td>B6 1</td>
<td>DC 81 B2 14</td>
<td>00 0</td>
<td>B6 1F 7B 44</td>
<td>74 0</td>
<td>10 F5 B8 2E</td>
</tr>
<tr>
<td>3</td>
<td>BE 0</td>
<td>2F 7E 3D A8</td>
<td>C0 1</td>
<td>27 95 A4 74</td>
<td>C4 0</td>
<td>07 11 6B D8</td>
<td>BC 0</td>
<td>C7 B7 AF C2</td>
</tr>
<tr>
<td>4</td>
<td>7E 1</td>
<td>32 21 1C 2C</td>
<td>8A 1</td>
<td>22 C2 DC 34</td>
<td>BC 1</td>
<td>BA DD 37 D8</td>
<td>DC 0</td>
<td>E7 42 39 BA</td>
</tr>
<tr>
<td>5</td>
<td>98 0</td>
<td>A9 76 2B EE</td>
<td>54 0</td>
<td>BC 91 D5 92</td>
<td>98 1</td>
<td>80 BA 9B F6</td>
<td>BC 1</td>
<td>48 16 81 0A</td>
</tr>
<tr>
<td>6</td>
<td>38 0</td>
<td>5D 4D F7 DA</td>
<td>BC 1</td>
<td>69 C2 8C 74</td>
<td>8A 1</td>
<td>A8 CE 7F DA</td>
<td>38 1</td>
<td>FA 93 EB 48</td>
</tr>
<tr>
<td>7</td>
<td>8A 1</td>
<td>04 2A 32 6A</td>
<td>9E 0</td>
<td>B1 86 56 0E</td>
<td>CC 1</td>
<td>96 30 47 F2</td>
<td>BC 1</td>
<td>F8 1D 42 30</td>
</tr>
</tbody>
</table>

A. What is size \((C)\) of this cache in bytes?

B. The box that follows shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

- \(CO\) The cache block offset
- \(CI\) The cache set index
- \(CT\) The cache tag

```
 1 2 3 4 5 6 7 8 9 10 11 12
```

6.32 Suppose that a program using the cache in Problem 6.31 references the 1-byte word at address \(0x071A\). Indicate the cache entry accessed and the cache byte value returned in \textbf{hex}. Indicate whether a cache miss occurs. If there is a cache miss, enter “–” for “Cache byte returned”. \textit{Hint: Pay attention to those valid bits!}

A. Address format (one bit per box):

```
 1 2 3 4 5 6 7 8 9 10 11 12
```
B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block offset (CO)</td>
<td>0x____</td>
</tr>
<tr>
<td>Index (CI)</td>
<td>0x____</td>
</tr>
<tr>
<td>Cache tag (CT)</td>
<td>0x____</td>
</tr>
<tr>
<td>Cache hit? (Y/N)</td>
<td>______</td>
</tr>
<tr>
<td>Cache byte returned</td>
<td>0x____</td>
</tr>
</tbody>
</table>

6.33 ◆◆
Repeat Problem 6.32 for memory address 0x16E8.

A. Address format (one bit per box):

```
  12  11  10  09  08  07  06  05  04  03  02  01  00
```

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache offset (CO)</td>
<td>0x____</td>
</tr>
<tr>
<td>Cache index (CI)</td>
<td>0x____</td>
</tr>
<tr>
<td>Cache tag (CT)</td>
<td>0x____</td>
</tr>
<tr>
<td>Cache hit? (Y/N)</td>
<td>______</td>
</tr>
<tr>
<td>Cache byte returned</td>
<td>0x____</td>
</tr>
</tbody>
</table>

6.34 ◆◆
For the cache in Problem 6.31, list the eight memory addresses (in hex) that will hit in set 2.

6.35 ◆◆
Consider the following matrix transpose routine:

```c
1  typedef int array[4][4];
2  
3  void transpose2(array dst, array src)
4  {
5      int i, j;
6  
7      for (i = 0; i < 4; i++) {
8          for (j = 0; j < 4; j++) {
9              dst[j][i] = src[i][j];
10         }
11     }
12  }
```
Assume this code runs on a machine with the following properties:

- \texttt{sizeof(int) == 4}.
- The \texttt{src} array starts at address 0 and the \texttt{dst} array starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate, with a block size of 16 bytes.
- The cache has a total size of 32 data bytes and the cache is initially empty.
- Accesses to the \texttt{src} and \texttt{dst} arrays are the only sources of read and write misses, respectively.

A. For each row and col, indicate whether the access to \texttt{src[row][col]} and \texttt{dst[row][col]} is a hit (h) or a miss (m). For example, reading \texttt{src[0][0]} is a miss and writing \texttt{dst[0][0]} is also a miss.

<table>
<thead>
<tr>
<th></th>
<th>dst array</th>
<th>src array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.36 ✭✭
Repeat Problem 6.35 for a cache with a total size of 128 data bytes.

<table>
<thead>
<tr>
<th></th>
<th>dst array</th>
<th>src array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.37 ✭✭
This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```c
int x[2][128];
int i;
int sum = 0;
for (i = 0; i < 128; i++) {
    sum += x[0][i] * x[1][i];
}
```
Assume we execute this under the following conditions:

- `sizeof(int) = 4.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:

A. Case 1: Assume the cache is 512 bytes, direct-mapped, with 16-byte cache blocks. What is the miss rate?
B. Case 2: What is the miss rate if we double the cache size to 1024 bytes?
C. Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate?
D. For Case 3, will a larger cache size help to reduce the miss rate? Why or why not?
E. For Case 3, will a larger block size help to reduce the miss rate? Why or why not?

6.38 ��
This is another problem that tests your ability to analyze the cache behavior of C code. Assume we execute the three summation functions in Figure 6.49 under the following conditions:

- `sizeof(int) == 4.
- The machine has a 4KB direct-mapped cache with a 16-byte block size.
- Within the two loops, the code uses memory accesses only for the array data. The loop indices and the value `sum` are held in registers.
- Array `a` is stored starting at memory address `0x08000000`.

Fill in the table for the approximate cache miss rate for the two cases `N = 64` and `N = 60`.

<table>
<thead>
<tr>
<th>Function</th>
<th><code>N = 64</code></th>
<th><code>N = 60</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sumA</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sumB</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sumC</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.39 ��
3M\textsuperscript{TM} decides to make Post-It\textsuperscript{®} notes by printing yellow squares on white pieces of paper. As part of the printing process, they need to set the CMYK (cyan, magenta, yellow, black) value for every point in the square. 3M hires you to determine
typedef int array_t[N][N];

int sumA(array_t a)
{
  int i, j;
  int sum = 0;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
      sum += a[i][j];
    }
  return sum;
}

int sumB(array_t a)
{
  int i, j;
  int sum = 0;
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++) {
      sum += a[i][j];
    }
  return sum;
}

int sumC(array_t a)
{
  int i, j;
  int sum = 0;
  for (j = 0; j < N; j+=2)
    for (i = 0; i < N; i+=2) {
      sum += (a[i][j] + a[i+1][j]
        + a[i][j+1] + a[i+1][j+1]);
    }
  return sum;
}

Figure 6.49 Functions referenced in Problem 6.38.

The efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks. You are given the following definitions:

struct point_color {
  int c;
  int m;
  int y;
  int k;
};
struct point_color square[16][16];
int i, j;

Assume the following:

- sizeof(int) == 4.
- square begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array square. Variables i and j are stored in registers.

Determine the cache performance of the following code:

```c
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].y = 1;
        square[i][j].k = 0;
    }
}
```

A. What is the total number of writes?
B. What is the total number of writes that miss in the cache?
C. What is the miss rate?

6.40 ◆
Given the assumptions in Problem 6.39, determine the cache performance of the following code:

```c
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[j][i].c = 0;
        square[j][i].m = 0;
        square[j][i].y = 1;
        square[j][i].k = 0;
    }
}
```

A. What is the total number of writes?
B. What is the total number of writes that miss in the cache?
C. What is the miss rate?
6.41 ◆
Given the assumptions in Problem 6.39, determine the cache performance of the following code:

```c
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[i][j].y = 1;
    }
}
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].k = 0;
    }
}
```

A. What is the total number of writes?
B. What is the total number of writes that miss in the cache?
C. What is the miss rate?

6.42 ◆◆
You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640 × 480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines. The C structures you are using are as follows:

```c
struct pixel {
    char r;
    char g;
    char b;
    char a;
};

struct pixel buffer[480][640];
int i, j;
char *cptr;
int *iptr;
```

Assume the following:

- `sizeof(char) == 1` and `sizeof(int) == 4`.
- `buffer` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.
What percentage of writes in the following code will miss in the cache?

```
1 for (j = 0; j < 640; j++) {
2    for (i = 0; i < 480; i++){
3        buffer[i][j].r = 0;
4        buffer[i][j].g = 0;
5        buffer[i][j].b = 0;
6        buffer[i][j].a = 0;
7    }
8 }
```

6.43  ◆◆
Given the assumptions in Problem 6.42, what percentage of writes in the following code will miss in the cache?

```
1 char *cptr = (char *) buffer;
2 for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
3    *cptr = 0;
```

6.44  ◆◆
Given the assumptions in Problem 6.42, what percentage of writes in the following code will miss in the cache?

```
1 int *iptr = (int *)buffer;
2 for (; iptr < ((int *)buffer + 640*480); iptr++)
3    *iptr = 0;
```

6.45  ◆◆◆
Download the mountain program from the CS:APP2 Web site and run it on your favorite PC/Linux system. Use the results to estimate the sizes of the caches on your system.

6.46  ◆◆◆◆
In this assignment, you will apply the concepts you learned in Chapters 5 and 6 to the problem of optimizing code for a memory-intensive application. Consider a procedure to copy and transpose the elements of an \( N \times N \) matrix of type int. That is, for source matrix \( S \) and destination matrix \( D \), we want to copy each element \( s_{i,j} \) to \( d_{j,i} \). This code can be written with a simple loop,

```
1 void transpose(int *dst, int *src, int dim)
2 {
3    int i, j;
4
5    for (i = 0; i < dim; i++)
6        for (j = 0; j < dim; j++)
7            dst[j*dim + i] = src[i*dim + j];
8 }
```
where the arguments to the procedure are pointers to the destination (dst) and source (src) matrices, as well as the matrix size $N$ (dim). Your job is to devise a transpose routine that runs as fast as possible.

6.47 ◆◆◆◆
This assignment is an intriguing variation of Problem 6.46. Consider the problem of converting a directed graph $g$ into its undirected counterpart $g'$. The graph $g'$ has an edge from vertex $u$ to vertex $v$ if and only if there is an edge from $u$ to $v$ or from $v$ to $u$ in the original graph $g$. The graph $g$ is represented by its adjacency matrix $G$ as follows. If $N$ is the number of vertices in $g$, then $G$ is an $N \times N$ matrix and its entries are all either 0 or 1. Suppose the vertices of $g$ are named $v_0, v_1, v_2, \ldots, v_{N-1}$. Then $G[i][j]$ is 1 if there is an edge from $v_i$ to $v_j$ and is 0 otherwise. Observe that the elements on the diagonal of an adjacency matrix are always 1 and that the adjacency matrix of an undirected graph is symmetric. This code can be written with a simple loop:

```c
void col_convert(int *G, int dim) {
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
}
```

Your job is to devise a conversion routine that runs as fast as possible. As before, you will need to apply concepts you learned in Chapters 5 and 6 to come up with a good solution.

Solutions to Practice Problems

Solution to Problem 6.1 (page 565)
The idea here is to minimize the number of address bits by minimizing the aspect ratio $\max(r, c)/\min(r, c)$. In other words, the squarer the array, the fewer the address bits.

<table>
<thead>
<tr>
<th>Organization</th>
<th>$r$</th>
<th>$c$</th>
<th>$b_r$</th>
<th>$b_c$</th>
<th>$\max(b_r, b_c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 x 1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>16 x 4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>128 x 8</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>512 x 4</td>
<td>32</td>
<td>16</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1024 x 4</td>
<td>32</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Solution to Problem 6.2 (page 573)
The point of this little drill is to make sure you understand the relationship between cylinders and tracks. Once you have that straight, just plug and chug:
Disk capacity = \(\frac{512 \text{ bytes}}{\text{sector}} \times \frac{400 \text{ sectors}}{\text{track}} \times \frac{10,000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{2 \text{ platters}}{\text{disk}}\) 

= 8,192,000,000 bytes 
= 8.192 GB

**Solution to Problem 6.3 (page 575)**

This solution to this problem is a straightforward application of the formula for disk access time. The average rotational latency (in ms) is 

\[
T_{\text{avg rotation}} = \frac{1}{2} \times T_{\text{max rotation}} \\
= \frac{1}{2} \times \left(\frac{60 \text{ secs}}{15,000 \text{ RPM}}\right) \times 1000 \text{ ms/sec} \\
\approx 2 \text{ ms}
\]

The average transfer time is 

\[
T_{\text{avg transfer}} = \left(\frac{60 \text{ secs}}{15,000 \text{ RPM}}\right) \times \frac{1}{500} \text{ sectors/track} \times 1000 \text{ ms/sec} \\
\approx 0.008 \text{ ms}
\]

Putting it all together, the total estimated access time is 

\[
T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} \\
= 8 \text{ ms} + 2 \text{ ms} + 0.008 \text{ ms} \\
\approx 10 \text{ ms}
\]

**Solution to Problem 6.4 (page 576)**

This is a good check of your understanding of the factors that affect disk performance. First we need to determine a few basic properties of the file and the disk. The file consists of 2000, 512-byte logical blocks. For the disk, \(T_{\text{avg seek}} = 5 \text{ ms}, T_{\text{max rotation}} = 6 \text{ ms}, \) and \(T_{\text{avg rotation}} = 3 \text{ ms}.\)

A. **Best case:** In the optimal case, the blocks are mapped to contiguous sectors, on the same cylinder, that can be read one after the other without moving the head. Once the head is positioned over the first sector it takes two full rotations (1000 sectors per rotation) of the disk to read all 2000 blocks. So the total time to read the file is \(T_{\text{avg seek}} + T_{\text{avg rotation}} + 2 \times T_{\text{max rotation}} = 5 + 3 + 12 = 20 \text{ ms.}\)

B. **Random case:** In this case, where blocks are mapped randomly to sectors, reading each of the 2000 blocks requires \(T_{\text{avg seek}} + T_{\text{avg rotation}} \text{ ms,}\) so the total time to read the file is \((T_{\text{avg seek}} + T_{\text{avg rotation}}) \times 2000 = 16,000 \text{ ms} (16 \text{ seconds}!)).

You can see now why it’s often a good idea to defragment your disk drive!
Solution to Problem 6.5 (page 581)
This problem, based on the zone map in Figure 6.14, is a good test of your understanding of disk geometry, and it also enables you to derive an interesting characteristic of a real disk drive.

A. Zone 0. There are a total of $864 \times 8 \times 3201 = 22,125,312$ sectors and $22,076,928$ logical blocks assigned to zone 0, for a total of $22,076,928 - 22,076,928 = 48,384$ spare sectors. Given that there are $864 \times 8 = 6912$ sectors per cylinder, there are $48,384/6912 = 7$ spare cylinders in zone 0.

B. Zone 8. A similar analysis reveals there are $(3700 \times 5632) - 20,804,608)/5632 = 6$ spare cylinders in zone 8.

Solution to Problem 6.6 (page 583)
This is a simple problem that will give you some interesting insights into feasibility of SSDs. Recall that for disks, 1 PB = $10^9$ MB. Then the following straightforward translation of units yields the following predicted times for each case:

A. Worst case sequential writes (170 MB/s): $10^9 \times (1/170) \times (1/(86,400 \times 365)) \approx 0.2$ years.

B. Worst case random writes (14 MB/s): $10^9 \times (1/14) \times (1/(86,400 \times 365)) \approx 2.25$ years.

C. Average case (20 GB/day): $10^9 \times (1/20,000) \times (1/365) \approx 140$ years.

Solution to Problem 6.7 (page 586)
In the 10-year period between 2000 and 2010, the unit price of rotating disk dropped by a factor of about 30, which means the price is dropping by roughly a factor of 2 every 2 years. Assuming this trend continues, a petabyte of storage, which costs about $300,000 in 2010, will drop below $500 after about ten of these factor-of-2 reductions. Since these are occurring every 2 years, we can expect a petabyte of storage to be available for $500 around the year 2030.

Solution to Problem 6.8 (page 590)
To create a stride-1 reference pattern, the loops must be permuted so that the rightmost indices change most rapidly.

```c
1   int sumarray3d(int a[N][N][N])
2   {
3       int i, j, k, sum = 0;
4
5       for (k = 0; k < N; k++) {
6           for (i = 0; i < N; i++) {
7               for (j = 0; j < N; j++) {
8                   sum += a[k][i][j];
9               }
10           }
11       }
12       return sum;
13   }
```
This is an important idea. Make sure you understand why this particular loop permutation results in a stride-1 access pattern.

**Solution to Problem 6.9 (page 590)**
The key to solving this problem is to visualize how the array is laid out in memory and then analyze the reference patterns. Function clear1 accesses the array using a stride-1 reference pattern and thus clearly has the best spatial locality. Function clear2 scans each of the $N$ structs in order, which is good, but within each struct it hops around in a non-stride-1 pattern at the following offsets from the beginning of the struct: 0, 12, 4, 16, 8, 20. So clear2 has worse spatial locality than clear1. Function clear3 not only hops around within each struct, but it also hops from struct to struct. So clear3 exhibits worse spatial locality than clear2 and clear1.

**Solution to Problem 6.10 (page 598)**
The solution is a straightforward application of the definitions of the various cache parameters in Figure 6.28. Not very exciting, but you need to understand how the cache organization induces these partitions in the address bits before you can really understand how caches work.

<table>
<thead>
<tr>
<th>Cache</th>
<th>$m$</th>
<th>$C$</th>
<th>$B$</th>
<th>$E$</th>
<th>$S$</th>
<th>$t$</th>
<th>$s$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>1</td>
<td>256</td>
<td>22</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>4</td>
<td>32</td>
<td>24</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>32</td>
<td>1</td>
<td>27</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

**Solution to Problem 6.11 (page 605)**
The padding eliminates the conflict misses. Thus, three-fourths of the references are hits.

**Solution to Problem 6.12 (page 605)**
Sometimes, understanding why something is a bad idea helps you understand why the alternative is a good idea. Here, the bad idea we are looking at is indexing the cache with the high-order bits instead of the middle bits.

A. With high-order bit indexing, each contiguous array chunk consists of $2^t$ blocks, where $t$ is the number of tag bits. Thus, the first $2^t$ contiguous blocks of the array would map to set 0, the next $2^t$ blocks would map to set 1, and so on.

B. For a direct-mapped cache where $(S, E, B, m) = (512, 1, 32, 32)$, the cache capacity is 512 32-byte blocks, and there are $t = 18$ tag bits in each cache line. Thus, the first $2^{18}$ blocks in the array would map to set 0, the next $2^{18}$ blocks to set 1. Since our array consists of only $(4096 * 4)/32 = 512$ blocks, all of the blocks in the array map to set 0. Thus, the cache will hold at most one array block at any point in time, even though the array is small enough to fit...
entirely in the cache. Clearly, using high-order bit indexing makes poor use of the cache.

**Solution to Problem 6.13 (page 609)**
The 2 low-order bits are the block offset (CO), followed by 3 bits of set index (CI), with the remaining bits serving as the tag (CT):

```
12 11 10 9 8 7 6 5 4 3 2 1 0
| CT | CT | CT | CT | CT | CT | CT | CI | CI | CI | CO | CO |
```

**Solution to Problem 6.14 (page 610)**
Address: 0x0E34

A. Address format (one bit per box):

```
12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| CT | CT | CT | CT | CT | CT | CT | CI | CI | CI | CO | CO |
```

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache block offset (CO)</td>
<td>0x0</td>
</tr>
<tr>
<td>Cache set index (CI)</td>
<td>0x5</td>
</tr>
<tr>
<td>Cache tag (CT)</td>
<td>0x71</td>
</tr>
<tr>
<td>Cache hit? (Y/N)</td>
<td>Y</td>
</tr>
<tr>
<td>Cache byte returned</td>
<td>0xB</td>
</tr>
</tbody>
</table>

**Solution to Problem 6.15 (page 611)**
Address: 0x0DD5

A. Address format (one bit per box):

```
12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| CT | CT | CT | CT | CT | CT | CT | CI | CI | CI | CO | CO |
```

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache block offset (CO)</td>
<td>0x1</td>
</tr>
<tr>
<td>Cache set index (CI)</td>
<td>0x5</td>
</tr>
<tr>
<td>Cache tag (CT)</td>
<td>0x6E</td>
</tr>
<tr>
<td>Cache hit? (Y/N)</td>
<td>N</td>
</tr>
<tr>
<td>Cache byte returned</td>
<td>—</td>
</tr>
</tbody>
</table>
Solutions to Practice Problems

Solution to Problem 6.16 (page 611)
Address: 0x1FE4

A. Address format (one bit per box):

```
   12 11 10  9  8  7  6  5  4  3  2  1  0
   CT CT CT CT CT CT CT CT CI CI CI CI CI CO CO
```

B. Memory reference:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache block offset</td>
<td>0x0</td>
</tr>
<tr>
<td>Cache set index</td>
<td>0x1</td>
</tr>
<tr>
<td>Cache tag</td>
<td>0xFF</td>
</tr>
<tr>
<td>Cache hit? (Y/N)</td>
<td>N</td>
</tr>
<tr>
<td>Cache byte returned</td>
<td>—</td>
</tr>
</tbody>
</table>

Solution to Problem 6.17 (page 611)

This problem is a sort of inverse version of Problems 6.13–6.16 that requires you to work backward from the contents of the cache to derive the addresses that will hit in a particular set. In this case, set 3 contains one valid line with a tag of 0x32. Since there is only one valid line in the set, four addresses will hit. These addresses have the binary form 0 0110 0100 11xx. Thus, the four hex addresses that hit in set 3 are 0x064C, 0x064D, 0x064E, and 0x064F.

Solution to Problem 6.18 (page 618)

A. The key to solving this problem is to visualize the picture in Figure 6.50. Notice that each cache line holds exactly one row of the array, that the cache is exactly large enough to hold one array, and that for all $i$, row $i$ of $src$ and $dst$ maps to the same cache line. Because the cache is too small to hold both arrays, references to one array keep evicting useful lines from the other array. For example, the write to $dst[0][0]$ evicts the line that was loaded when we read $src[0][0]$. So when we next read $src[0][1]$, we have a miss.

![Figure 6.50](image-url)

**Figure 6.50**
**Figure for Problem 6.18.**
B. When the cache is 32 bytes, it is large enough to hold both arrays. Thus, the only misses are the initial cold misses.

<table>
<thead>
<tr>
<th></th>
<th>dst array</th>
<th>src array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>Row 1</td>
<td>m</td>
<td>m</td>
</tr>
</tbody>
</table>

**Solution to Problem 6.19 (page 619)**

Each 16-byte cache line holds two contiguous *algae_position* structures. Each loop visits these structures in memory order, reading one integer element each time. So the pattern for each loop is miss, hit, miss, hit, and so on. Notice that for this problem we could have predicted the miss rate without actually enumerating the total number of reads and misses.

A. What is the total number of read accesses? 512 reads.
B. What is the total number of read accesses that miss in the cache? 256 misses.
C. What is the miss rate? $\frac{256}{512} = 50\%$.

**Solution to Problem 6.20 (page 620)**

The key to this problem is noticing that the cache can only hold 1/2 of the array. So the column-wise scan of the second half of the array evicts the lines that were loaded during the scan of the first half. For example, reading the first element of *grid*[8][0] evicts the line that was loaded when we read elements from *grid*[0][0]. This line also contained *grid*[0][1]. So when we begin scanning the next column, the reference to the first element of *grid*[0][1] misses.

A. What is the total number of read accesses? 512 reads.
B. What is the total number of read accesses that miss in the cache? 256 misses.
C. What is the miss rate? $\frac{256}{512} = 50\%$.
D. What would the miss rate be if the cache were twice as big? If the cache were twice as big, it could hold the entire *grid* array. The only misses would be the initial cold misses, and the miss rate would be $\frac{1}{4} = 25\%$.

**Solution to Problem 6.21 (page 620)**

This loop has a nice stride-1 reference pattern, and thus the only misses are the initial cold misses.

A. What is the total number of read accesses? 512 reads.
B. What is the total number of read accesses that miss in the cache? 128 misses.
C. What is the miss rate? $\frac{128}{512} = 25\%$. 
D. What would the miss rate be if the cache were twice as big? Increasing the cache size by any amount would not change the miss rate, since cold misses are unavoidable.

**Solution to Problem 6.22 (page 625)**
The peak throughput from L1 is about 6500 MB/s, the clock frequency is 2670 MHz, and the individual read accesses are in units of 8-byte doubles. Thus, from this graph we can estimate that it takes roughly $\frac{2670}{6500} \times 8 = 3.2 \approx 4$ cycles to access a word from L1 on this machine.