Chapter 5- Optimizing Performance

Writing an efficient program requires several types of activities such as :

- must select an appropriate set of algorithms and data structures to solve the problem
- must write source code that the compiler can effectively optimize
- must understand the capabilities and limitations of optimizing compilers
- divide a task into parts that can be computed in parallel, multiple cores and multiple processors.
- code that is executed repeatedly in a critical environment, may require optimization
- eliminate unnecessary work, which includes eliminating unnecessary function calls, conditional tests, and memory references.
- both the programmer and compiler require a target machine, about how instructions are processed and the timing characteristics of the different operations.
- use the capability of processors to provide instruction-level parallelism, executing multiple instructions simultaneously
- use code profilers —tools that measure the performance - of different parts of a program to detect inefficiency in the code and improve them
- assembly-code must be understood in the way it does inner loops, identifying performance-reducing attributes such as excessive memory references and poor use of registers
- identify critical paths , chains of data dependencies over repeated executions of a loop
- make the compiler generate efficient code without compromising the readability, modularity, and portability of the code

Capabilities and Limitations of Optimizing Compilers

- specify the optimization level by invoking gcc with the command-line flag '-O1' will cause it to apply a basic set of optimizations
- invoking gcc with flag '-O2'or'-O3' will cause it to apply more extensive optimizations
- more extensive optimizations may expand the code size and may make the program more difficult to debug
- C code when compiled just with optimization level 1, vastly outperforms a naive version compiled with the highest possible optimization levels.
- make the compiler perform only safe optimizations to eliminate possible sources of undesired run-time behavior

```
1   void twiddle1(int *xp, int *yp)
2   {
3       *xp += *yp;
4       *xp += *yp;
5   }
6
7   void twiddle2(int *xp, int *yp)
8   {
9       *xp += 2* *yp;
10  }
```

Function twiddle2 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). So twiddle2 is more efficient than twiddle1.

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

```
3       *xp += *xp;  /* Double value at xp */
4       *xp += *xp;  /* Double value at xp */
```

The result will be that the value at xp will be increased by a factor of 4. On the other hand, function twiddle2 will perform the following computation:

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

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

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

```
x = 1000; y = 3000;
*q = y;   /* 3000 */
*p = x;   /* 1000 */
t1 = *q;  /* 1000 or 3000 */
```

The value computed for t1 depends on whether or not pointers p and q are aliased—if not, it will equal 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.

A second optimization blocker is due to function calls. As an example, consider the following two procedures:

```
1   int f();
2
3   int func1() {
4       return f() + f() + f() + f();
5   }
6
7   int func2() {
8       return 4*f();
9   }
```

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

Consider, however, the following code for f:

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

This function has a *side effect*—it modifies some part of the global program state. Changing the number of times it gets called changes the program behavior. In particular, a call to func1 would return $0+1+2+3=6$, whereas a call to func2 would return $4 \cdot 0 = 0$, assuming both started with global variable counter set to 0.

Most compilers do not try to determine whether a function is free of side effects and hence is a candidate for optimizations such as those attempted in func2. Instead, the compiler assumes the worst case and leaves function calls intact.

A second optimization blocker is due to function calls. As an example, consider the following two procedures:

```
1  int f();
2
3  int func1() {
4      return f() + f() + f() + f();
5  }
6
7  int func2() {
8      return 4*f();
9  }
```

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

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 func1 by substituting four instantiations of function f:

```
1  /* Result of inlining f in func1 */
2  int func1in() {
3      int t = counter++;  /* +0 */
4      t += counter++;     /* +1 */
5      t += counter++;     /* +2 */
6      t += counter++;     /* +3 */
7      return t;
8  }
```

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

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

This code faithfully reproduces the behavior of func1 for this particular definition of function f.

Recent versions of GCC attempt this form of optimization, either when directed to with the command-line option '-finline' or for optimization 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.

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 4.0×10^9 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.

Non-Pipelined and Pipelining exercise

The 5 stages (Fetch, Decode, Execute, Memory, Writeback) of the processor have the following latencies:

	Fetch	Decode	Execute	Memory	Writeback
	300	300	375	500	125

Assume that when pipelining, each pipeline stage costs 20 ps extra for the registers between pipeline stages.

Non-pipelined processor case:
What is the cycle time? (**CT)**
What is the latency of an instruction? (**Latency)**
What is the throughput? (**Throughput)**

Because there is no pipelining, the cycle time must allow an instruction to go through all stages in one cycle. The latency is the same as cycle time since it takes the instruction one cycle to go from the beginning of fetch to the end of writeback. The Throughput is defined as 1/CT inst/ps. (ps – picoseconds)

CT = 300 + 300 + 375 + 500 + 125 = 1600 ps
Latency = 1600 ps
Throughput = 1/CT = 1/1600 inst/ps

Pipelined processor case:
What is the cycle time? **(CT)**
What is the latency of an instruction? (**Latency)**
What is the throughput? (**Throughput**)

Pipelining reduces the cycle time to the length of the longest stage plus the register delay. Latency becomes CT*N (CT-Cycle time) where N is the number of stages as one instruction will need to go through each of the stages and each stage takes one cycle. The throughput formula remains the same as 1/CT instr/ps.

CT = 500 + 20 = 520 ps
Latency = 5 * 520 = 2600 ps
Throughput = 1/520 inst/ps

Splitting a pipeline stage case:

	Fetch	Decode	Execute	Memory	Writeback
	300	300	375	500	125

If you could split one of the pipeline stages into 2 equal halves, which one would you choose?
Assume that when pipelining, each pipeline stage costs 20 ps extra for the registers between pipeline stages.

What is the new cycle time?
What is the new latency?
What is the new throughput?

We would want to choose the longest stage to split in half- the 500 one. The new cycle time becomes the originally 2nd longest stage length - 375. Calculate latency and throughput correspondingly, but remember there are now 6 stages instead of 5.

CT = 375 + 20 = 395 ps
Latency = 6 * 395 = 2370 ps
Throughput = 1/395 inst/ps

You are given a non-pipelined processor design which has a cycle time of 10ns and average CPI of 1.4.

If the 5 stages are 1ns, 1.5ns, 4ns, 3ns, and 0.5ns, what is the best speedup you can get compared to the original processor?
The cycle time is limited by the slowest stage, so CT = 4ns.
Speedup = old CT / new CT = 10ns/4ns = 2.5x

Many procedures contain a loop that iterates over a set of elements. For example, functions psum1 and psum2 in Figure 5.1 both compute the *prefix sum* of a vector of length n. For a vector $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, the prefix sum $\vec{p} = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ is defined as

$$p_0 = a_0$$
$$p_i = p_{i-1} + a_i, 1 \leq i < n \qquad (5.1)$$

Function psum1 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 psum1 and psum2 can be approximated by the equations $496 + 10.0n$ and $500 + 6.5n$, respectively. These equations indicate an overhead of 496 to 500

```
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.

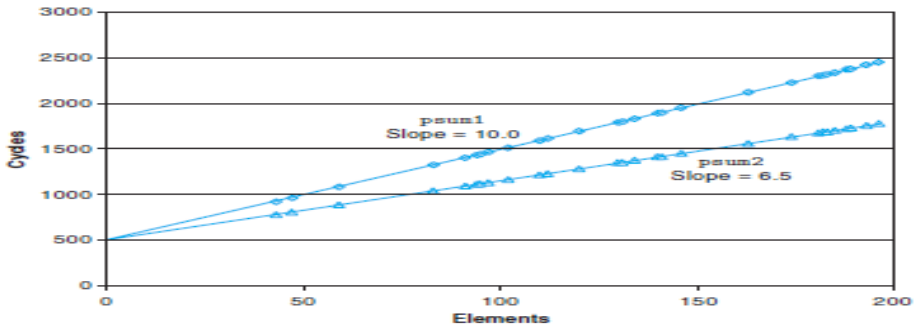


Figure 5.2 **Performance of prefix-sum functions.** The slope of the lines indicates the

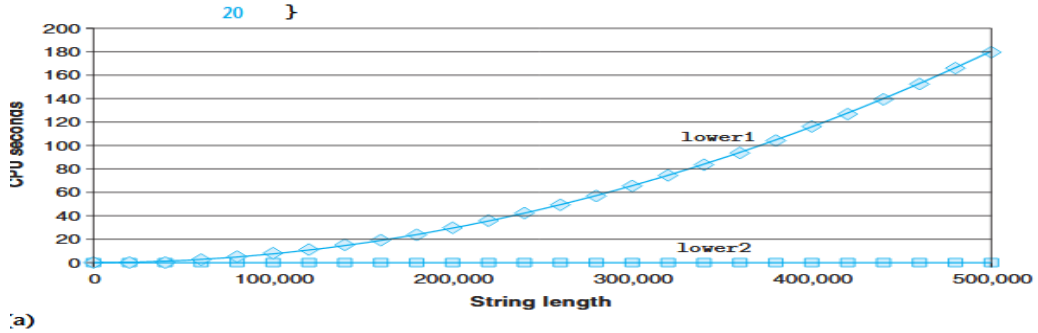
Function lower2 shown in Figure 5.7 is identical to that of lower1 , except that we have moved the call to Strlen out of the loop.

```
/* Convert string to lowercase: slow */
void lower1(char *s)
{
    int i;

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

/* Convert string to lowercase: faster */
void lower2(char *s)
{
    int i;
    int len = strlen(s);

    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```



(a)

	String length						
Function	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
lower1	0.19	0.77	3.08	12.34	49.39	198.42	791.22
lower2	0.0000	0.0000	0.0001	0.0002	0.0004	0.0008	0.0015

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

```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6
7       *dest = IDENT;
8       for (i = 0; i < length; i++) {
9           data_t val;
10          get_vec_element(v, i, &val);
11          *dest = *dest OP val;
12      }
13  }
```

Figure 5.6 **Improving the efficiency of the loop test.** By moving the call to `vec_length` out of the loop test, we eliminate the need to execute it on every iteration.

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.

			Integer		Floating point		
Function	Page	Method	+	*	+	F *	D *
combine1	485	Abstract -O1	12.00	12.00	12.00	12.01	13.00
combine2	486	Move vec_length	8.03	8.09	10.09	11.09	12.08

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.

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

```
1  /* Move call to vec_length out of loop */
2  void combine2(vec_ptr v, data_t *dest)
3  {
4      long int i;
5      long int length = vec_length(v);
6
7      *dest = IDENT;
8      for (i = 0; i < length; i++) {
9          data_t val;
10         get_vec_element(v, i, &val);
11         *dest = *dest OP val;
12     }
13 }
```

Figure 5.6 **Improving the efficiency of the loop test.** By moving the call to `vec_length` out of the loop test, we eliminate the need to execute it on every iteration.

code/opt/vec.c

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

code/opt/vec.c

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

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

contents are stored as an array, rather than as some other data structure such as a linked list. A more pragmatic programmer would argue that this transformation is a necessary step toward achieving high-performance results.

			Integer		Floating point		
Function	Page	Method	+	*	+	F *	D *
combine2	486	Move vec_length	8.03	8.09	10.09	11.09	12.08
combine3	491	Direct data access	6.01	8.01	10.01	11.01	12.02

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.

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

here the x86-64 code generated for data type float and with multiplication a 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
```

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.


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

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

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.

			Integer		Floating point		
Function	Page	Method	+	*	+	F *	D *
combine3	491	Direct data access	6.01	8.01	10.01	11.01	12.02
combine4	493	Accumulate in temporary	2.00	3.00	3.00	4.00	5.00

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

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.

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

Figure 5.16 **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:

			Integer		Floating point		
Function	Page	Method	+	*	+	F *	D *
combine4	493	No unrolling	2.00	3.00	3.00	4.00	5.00
combine5	510	Unroll by ×2	2.00	1.50	3.00	4.00	5.00
		Unroll by ×3	1.00	1.00	3.00	4.00	5.00
Latency bound			1.00	3.00	3.00	4.00	5.00
Throughput bound			1.00	1.00	1.00	1.00	1.00