By using multiple optimizations, we have been able to achieve CPEs close to the throughput bounds of 0.50 and 1.00, limited only by the capacities of the functional units. These represent 10–20× improvements on the original code. This has all been done using ordinary C code and a standard compiler. Rewriting the code to take advantage of the newer SIMD instructions yields additional performance gains of nearly 4× or 8×. For example, for single-precision multiplication, the CPE drops from the original value of 11.14 down to 0.06, an overall performance gain of over 180×. This example demonstrates that modern processors have considerable amounts of computing power, but we may need to coax this power out of them by writing our programs in very stylized ways.

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 C functional units capable of performing that operation, and that these units have an issue time of I. Then the program will require at least $N \cdot I/C$ 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. If a program has 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 in memory, typically by allocating space on the run-time stack. As an example, the following measurements compare the result of extending the multiple accumulator scheme of `combine6` to the cases of $k = 10$ and $k = 20$:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine6`	573					
		10 × 10 unrolling	0.55	1.00	1.01	0.52
		20 × 20 unrolling	0.83	1.03	1.02	0.68
Throughput bound			0.50	1.00	1.00	0.50

We can see that none of the CPEs improve with this increased unrolling, and some even get worse. Modern x86-64 processors have 16 integer registers and can make use of the 16 YMM registers to store floating-point data. Once the number of loop variables exceeds the number of available registers, the program must allocate some on the stack.

As an example, the following snippet of code shows how accumulator `acc0` is updated in the inner loop of the code with 10×10 unrolling:

```
Updating of accumulator acc0 in 10 x 10 urolling
vmulsd  (%rdx), %xmm0, %xmm0     acc0 *= data[i]
```

We can see that the accumulator is kept in register `%xmm0`, and so the program can simply read `data[i]` from memory and multiply it by this register.

The comparable part of the code for 20×20 unrolling has a much different form:

```
Updating of accumulator acc0 in 20 x 20 unrolling
vmovsd  40(%rsp), %xmm0
vmulsd  (%rdx), %xmm0, %xmm0
vmovsd  %xmm0, 40(%rsp)
```

The accumulator is kept as a local variable on the stack, at offset 40 from the stack pointer. The program must read both its value and the value of `data[i]` from memory, multiply them, and store the result back to memory.

Once a compiler must resort to register spilling, any advantage of maintaining multiple accumulators will most likely be lost. Fortunately, x86-64 has enough registers that most loops will become throughput limited before this occurs.

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, including all processors capable of executing x86-64 programs, have *conditional move* instructions. 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 4.5.7 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 19-cycle misprediction penalty we measured for the reference machine, 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 results 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:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine2`	545	Move `vec_length`	7.02	9.03	9.02	11.03
`combine3`	549	Direct data access	7.17	9.02	9.02	11.03

The CPE did not improve, even though the transformation eliminated two conditionals on each iteration that check whether the vector index is within bounds. For this function, the checks always succeed, 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.

```
/* Include bounds check in loop */
void combine4b(vec_ptr v, data_t *dest)
{
    long i;
    long 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:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine4	551	No bounds checking	1.27	3.01	3.01	5.01
combine4b	551	Bounds checking	2.02	3.01	3.01	5.01

The version with bounds checking is slightly slower for the case of integer addition, but it achieves the same performance for the other three cases. The performance of these cases is limited by the latencies of their respective combining operations. The additional computation required to perform bounds checking can take place in parallel with the combining operations. 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. 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:

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

Our measurements for this function show a CPE of around 13.5 for random data and 2.5–3.5 for predictable data, an indication of a misprediction penalty of around 20 cycles.

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:

```
/* Rearrange two vectors so that for each i, b[i] >= a[i] */
void minmax2(long a[], long b[], long n) {
    long i;
    for (i = 0; i < n; i++) {
        long min = a[i] < b[i] ? a[i] : b[i];
        long 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 4.0 regardless of whether the data are arbitrary or predictable. (We also examined the generated assembly code to make sure that it indeed uses 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 **(solution page 612)**

The traditional implementation of the merge step of mergesort requires three loops [98]:

```
1   void merge(long src1[], long src2[], long dest[], long n) {
2       long i1 = 0;
3       long i2 = 0;
4       long id = 0;
5       while (i1 < n && i2 < n) {
6           if (src1[i1] < src2[i2])
7               dest[id++] = src1[i1++];
8           else
9               dest[id++] = src2[i2++];
10      }
11      while (i1 < n)
12          dest[id++] = src1[i1++];
13      while (i2 < n)
14          dest[id++] = src2[i2++];
15  }
```

The branches caused by comparing variables `i1` and `i2` to `n` have good prediction performance—the only mispredictions occur when they first become false. The comparison between values `src1[i1]` and `src2[i2]` (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 15.0 when run on random data.

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 1,000 elements, requiring no more than 8,000 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.