```
5      addq    $8, %rdx                Increment data+i
6      cmpq    %rax, %rdx              Compare to data+length
7      jne     .L17                    If !=, goto loop
```

We see that, besides some reordering of instructions, the only difference is that the more optimized version does not contain the `vmovsd` 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. Either explain 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 1.25–5 clock cycles for each element to be computed. This is a considerable improvement over the original 9–11 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 recent Intel processors. These processors are described in the industry as being *superscalar*, which means they 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

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.

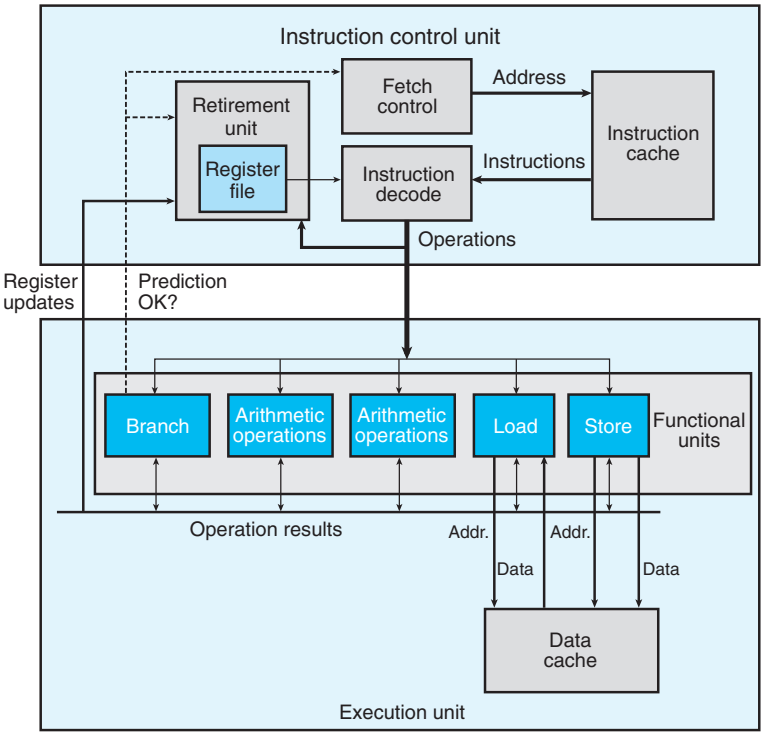

Figure 5.11 Block diagram of an out-of-order processor. The instruction control unit is responsible for reading instructions from memory and generating a sequence of primitive operations. The execution unit then performs the operations and indicates whether the branches were correctly predicted.

instruction 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 multiple operations. The details of how instructions are decoded into sequences of 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

```
addq %rax,%rdx
```

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

```
addq %rax,8(%rdx)
```

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 value in register `%eax`, and one to *store* the result back to memory. The 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 *functional units* that perform the actual operations. These functional units are specialized to handle different types of operations.

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 *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 functional units.

Figure 5.11 indicates that the different functional units are designed to perform different operations. Those labeled as performing "arithmetic operations" are typically specialized to perform different combinations of integer and floating-point operations. As the number of transistors that can be integrated onto a single

microprocessor chip has grown over time, successive models of microprocessors have increased the total number of functional units, the combinations of operations each unit can perform, and the performance of each of these units. The arithmetic units are intentionally designed to be able to perform a variety of different operations, since the required operations vary widely across different programs. For example, some programs might involve many integer operations, while others require many floating-point operations. If one functional unit were specialized to perform integer operations while another could only perform floating-point operations, then none of these programs would get the full benefit of having multiple functional units.

For example, our Intel Core i7 Haswell reference machine has eight functional units, numbered 0–7. Here is a partial list of each one's capabilities:

0. Integer arithmetic, floating-point multiplication, integer and floating-point division, branches

1. Integer arithmetic, floating-point addition, integer multiplication, floating-point multiplication

2. Load, address computation

3. Load, address computation

4. Store

5. Integer arithmetic

6. Integer arithmetic, branches

7. Store address computation

In the above list, "integer arithmetic" refers to basic operations, such as addition, bitwise operations, and shifting. Multiplication and division require more specialized resources. We see that a store operation requires two functional units—one to compute the store address and one to actually store the data. We will discuss the mechanics of store (and load) operations in Section 5.12.

We can see that this combination of functional units has the potential to perform multiple operations of the same type simultaneously. It has four units capable of performing integer operations, two that can perform load operations, and two that can perform floating-point multiplication. We will later see the impact these resources have on the maximum performance our programs can achieve.

Within the ICU, the *retirement unit* keeps track of the ongoing processing and makes sure that it obeys the sequential semantics of the machine-level program. Our figure shows a *register file* containing the integer, floating-point, and, more recently, SSE and AVX 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 *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

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 10 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 our reference machine.

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 4.5.5.

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.

	Integer			Floating point		
Operation	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

Figure 5.12 Latency, issue time, and capacity characteristics of reference machine 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 independent operations. The capacity indicates how many of these operations can be issued simultaneously. The times for division depend on the data values.

5.7.2 Functional Unit Performance

Figure 5.12 documents the performance of some of the arithmetic operations for our Intel Core i7 Haswell reference machine, determined by both measurements and by reference to Intel literature [49]. 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, the *issue time*, meaning the minimum number of clock cycles between two independent operations of the same type, and the *capacity*, indicating the number of functional units capable of performing that operation.

We see that the latencies increase in going from integer to floating-point operations. We see also that the addition and multiplication operations all 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. Operations with capacity greater than 1 arise due to the capabilities of the multiple functional units, as was described earlier for the reference machine.

We see also that the divider (used for integer and floating-point division, as well as floating-point square root) is not pipelined—its issue time equals its latency. What this means is that the divider must perform a complete division before it can begin a new one. We also see that 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 1 operation per clock cycle, while units with higher issue times have lower maximum throughput. Having multiple functional units can increase throughput even further. For an operation with capacity C and issue time I, the processor can potentially achieve a throughput of C/I operations per clock cycle. For example, our reference machine is capable of performing floating-point multiplication operations at a rate of 2 per clock cycle. We will see how this capability can be exploited to increase program performance.

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 the design of the Core i7 Haswell processor, 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.

The latencies, issue times, and capacities 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:

	Integer		Floating point	
Bound	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	0.50	1.00	1.00	0.50

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 integer multiplier, and it has an issue time of 1 clock cycle, the processor cannot possibly sustain a rate of more than 1 multiplication per clock cycle. On the other hand, with four functional units capable of performing integer addition, the processor can potentially sustain a rate of 4 operations per cycle. Unfortunately, the need to read elements from memory creates an additional throughput bound. The two load units limit the processor to reading at most 2 data values per clock cycle, yielding a throughput bound of 0.50. We will demonstrate the effect of both 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:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

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 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) 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 data type `double` with multiplication as the combining operation. Other combinations of data type and operation yield similar code. The compiled code for this loop consists of four instructions, with registers `%rdx` holding a pointer to the ith element of array `data`, `%rax` holding a pointer to the end of the array, and `%xmm0` holding the accumulated value `acc`.

```
    Inner loop of combine4.  data_t = double, OP = *
    acc in %xmm0, data+i in %rdx, data+length in %rax
1   .L25:                                   loop:
2     vmulsd  (%rdx), %xmm0, %xmm0            Multiply acc by data[i]
3     addq    $8, %rdx                        Increment data+i
4     cmpq    %rax, %rdx                      Compare to data+length
5     jne     .L25                            If !=, goto loop
```

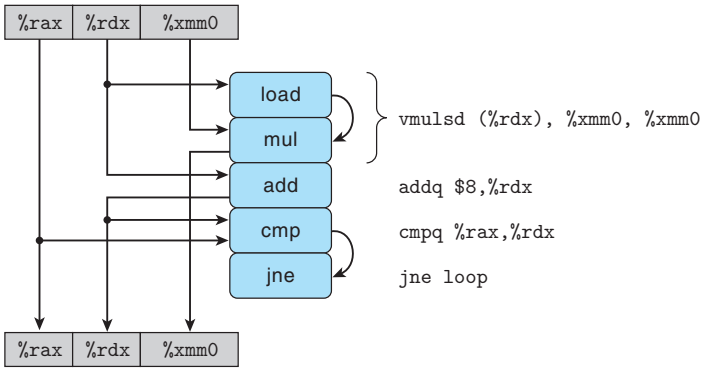

Figure 5.13 Graphical representation of inner-loop code for `combine4`. Instructions are dynamically translated into one or two operations, each of which receives values from other operations or from registers and produces values for other operations and for registers. We show the target of the final instruction as the label `loop`. It jumps to the first instruction shown.

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 cmp operation, and so the register has the same value at the end of the loop as at the beginning. 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 `vmulsd` 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 jne operation.

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

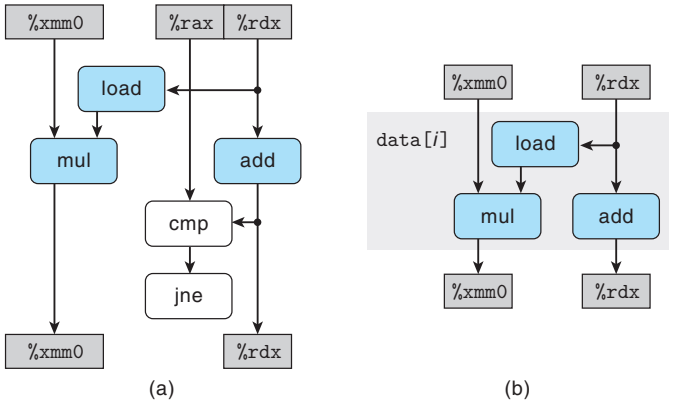

Figure 5.14
Abstracting `combine4` operations as a data-flow graph. We rearrange the operators of Figure 5.13 to more clearly show the data dependencies (a), and then further show only those operations that use values from one iteration to produce new values for the next (b).

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 only read-only register for the loop in `combine4` is `%rax`.

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

Loop. These are used both 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 `data+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 comparison (cmp) and branch (jne) 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 taken. 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 a data element, generated by the load operation. Along the other side, we see the dependencies between successive values of the pointer to the ith data element. On each iteration, the old value is used as the address for the load operation, and it is also incremented by the add operation to compute its new value.

Figure 5.15 shows the data-flow representation of n iterations by the inner loop of function `combine4`. This graph was obtained by simply replicating the template shown in Figure 5.14(b) n times. We can see that the program has two chains of data

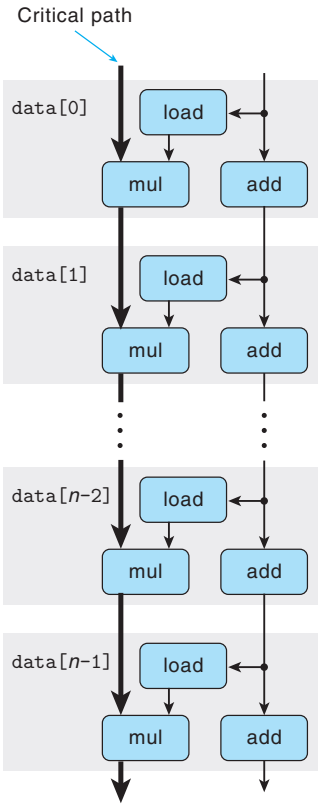

Figure 5.15
Data-flow representation of computation by n iterations of the inner loop of `combine4`. The sequence of multiplication operations forms a critical path that limits program performance.

dependencies, corresponding to the updating of program values `acc` and `data+i` with operations mul and add, respectively. Given that floating-point multiplication has a latency of 5 cycles, while integer addition has a latency of 1 cycle, we can see that the chain on the left will form a *critical path*, requiring $5n$ cycles to execute. The chain on the right would require only n cycles to execute, and so it does not limit the program performance.

Figure 5.15 demonstrates why we achieved a CPE equal to the latency bound of 5 cycles for `combine4`, when performing floating-point multiplication. When executing the function, the floating-point multiplier becomes the limiting resource. The other operations required during the loop—manipulating and testing pointer value `data+i` and reading data from memory—proceed in parallel with the multiplication. As each successive value of `acc` is computed, it is fed back around to compute the next value, but this will not occur until 5 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 1.27, 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 1.27 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 1.27 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 below or close to 1.00.

Practice Problem 5.5 (solution page 611)

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

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{5.2}$$

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` (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:

```
1  double poly(double a[], double x, long degree)
2  {
3      long 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 (solution page 611)

Let us continue exploring ways to evaluate polynomials, as described in Practice 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:

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)) \tag{5.3}$$

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

```
1  /* Apply Horner's method */
2  double polyh(double a[], double x, long degree)
3  {
```

```
4       long 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 Practice 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 what we will refer to as "2×1 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, yielding $k \times 1$ *loop unrolling*. 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