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.

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, our reference machine has two load units, each of which can hold up to 72 pending read requests. It has a single store unit with a store buffer containing up to 42 write requests. Each of these units can initiate 1 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 using our reference machine, we saw that the CPE never got below 0.50 for any combination of data type and combining operation, 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. With two load units, each able to initiate at most 1 load operation every clock cycle, the CPE cannot be less than 0.50. For applications where we must load k values for every element computed, we can never achieve a CPE lower than $k/2$ (see, for example, Problem 5.15).

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 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

```
typedef struct ELE {
    struct ELE *next;
    long data;
} list_ele, *list_ptr;

long list_len(list_ptr ls) {
    long len = 0;
    while (ls) {
      len++;
      ls = ls->next;
    }
    return len;
}
```

Figure 5.31 Linked list function. Its performance is limited by the latency of the load operation.

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:

```
     Inner loop of list_len
     ls in %rdi, len in %rax
1    .L3:                          loop:
2      addq    $1, %rax              Increment len
3      movq    (%rdi), %rdi          ls = ls->next
4      testq   %rdi, %rdi            Test ls
5      jne     .L3                   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. Indeed, this measurement matches the documented access time of 4 cycles for the reference machine's L1 cache, as is discussed in Section 6.4.

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 function shown in Figure 5.32 that sets the elements of an array dest of length n to zero. Our measurements show a CPE of 1.0. This is the best we can achieve on a machine with a single store functional unit.

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

```
/* Set elements of array to 0 */
void clear_array(long *dest, long n) {
    long i;
    for (i = 0; i < n; i++)
        dest[i] = 0;
}
```

Figure 5.32 **Function to set array elements to 0.** This code achieves a CPE of 1.0.

```
/* Write to dest, read from src */
void write_read(long *src, long *dst, long n)
{
    long cnt = n;
    long val = 0;

    while (cnt) {
        *dst = val;
        val = (*src)+1;
        cnt--;
    }
}
```

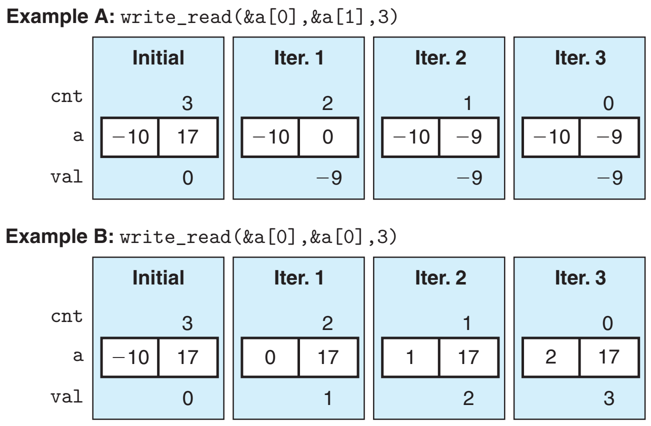

Figure 5.33 Code to write and read memory locations, along with illustrative executions. This function highlights the interactions between stores and loads when arguments `src` and `dest` are equal.

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 `cnt` equal to 3. These executions illustrate some subtleties of the load and store operations.

In Example A of Figure 5.33, argument `src` is a pointer to array element `a[0]`, while `dest` is a pointer to array element `a[1]`. In this case, each load by the pointer reference `*src` will yield the value −10. Hence, after two iterations, the array elements will remain fixed at −10 and −9, respectively. The result of the read from `src` is not affected by the write to `dest`. Measuring this example over a larger number of iterations gives a CPE of 1.3.

In Example B of Figure 5.33, both arguments `src` and `dest` are pointers to array element `a[0]`. In this case, each load by the pointer reference `*src` will yield the value stored by the previous execution of the pointer reference `*dest`.

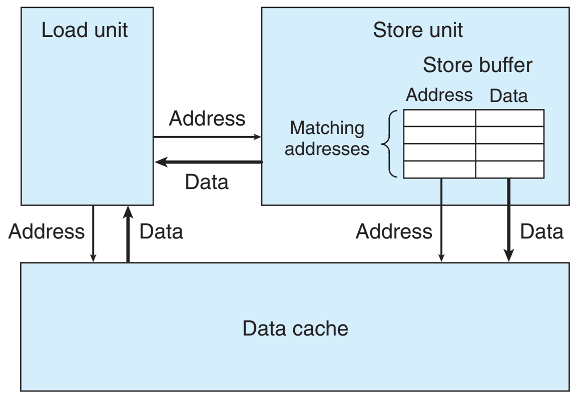

Figure 5.34
Detail of load and store units. The store unit maintains a buffer of pending writes. The load unit must check its address with those in the store unit to detect a write/read dependency.

As a consequence, a series of ascending values will be stored in this location. In general, if function `write_read` is called with arguments `src` and `dest` pointing to the same memory location, and with argument `cnt` having some value $n > 0$, the net effect is to set the location to $n - 1$. This example illustrates a phenomenon we will call 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 7.3. The write/read dependency causes a slowdown in the processing of around 6 clock cycles.

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 includes a *store buffer* containing the addresses and data of the store operations that have been issued to the store unit, but have not yet been completed, where completion involves updating the data cache. This buffer is provided so that a series of store operations can be executed without having to wait for each one to update the cache. When a load operation occurs, it must check the entries in the store buffer for matching addresses. If it finds a match (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.

GCC generates the following code for the inner loop of `write_read`:

```
  Inner loop of write_read
  src in %rdi, dst in %rsi, val in %rax
.L3:                          loop:
  movq    %rax, (%rsi)          Write val to dst
  movq    (%rdi), %rax          t = *src
  addq    $1, %rax              val = t+1
  subq    $1, %rdx              cnt--
  jne     .L3                   If != 0, goto loop
```

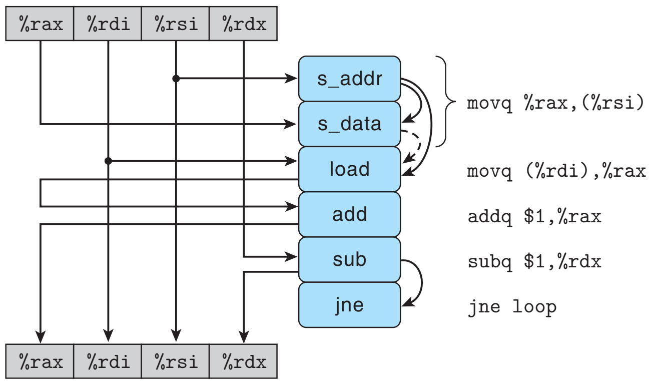

Figure 5.35
Graphical representation of inner-loop code for `write_read`. The first `movl` instruction is decoded into separate operations to compute the store address and to store the data to memory.

Figure 5.35 shows a data-flow representation of this loop code. The instruction `movq %rax,(%rsi)` 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. This motivates the separate functional units for these operations in the reference machine.

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 `movq (%rdi), %rax` 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 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`.

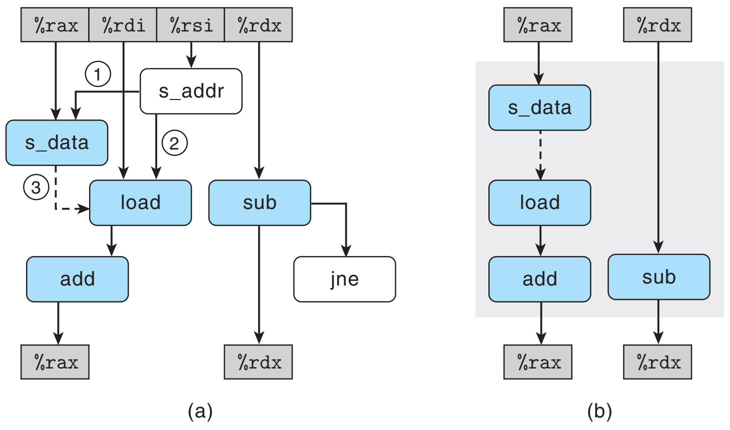

Figure 5.36
Abstracting the operations for `write_read`. We first rearrange the operators of Figure 5.35(a) and then show only those operations that use values from one iteration to produce new values for the next (b).

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 in 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`, resulting in a CPE bound of 1.0. For the case of Example B with matching source and destination addresses, the data dependency between the s_data and 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 around 7 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 (solution page 613)

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

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

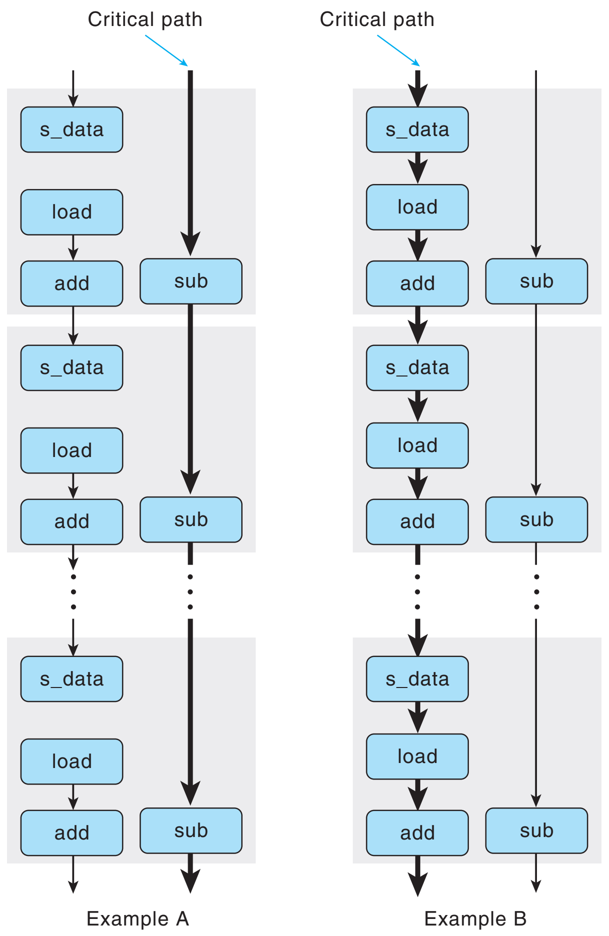

Figure 5.37
Data-flow representation of function `write_read`. When the two addresses do not match, the only critical path is formed by the decrementing of `cnt` (Example A). When they do match, the chain of data being stored, loaded, and incremented forms the critical path (Example B).

Suppose `a` is an array of length 1,000 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 1.2 (which drops to 1.0 when the loop is unrolled by a factor of 4), while the call of part B has a CPE of 5.0. 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 (solution page 613)

We saw that our measurements of the prefix-sum function `psum1` (Figure 5.1) yield a CPE of 9.00 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:

```
    Inner loop of psum1
    a in %rdi, i in %rax, cnt in %rdx
.L5:                                          loop:
  vmovss  -4(%rsi,%rax,4), %xmm0                Get p[i-1]
  vaddss  (%rdi,%rax,4), %xmm0, %xmm0           Add a[i]
  vmovss  %xmm0, (%rsi,%rax,4)                  Store at p[i]
  addq    $1, %rax                              Increment i
  cmpq    %rdx, %rax                            Compare i:cnt
  jne     .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.

Practice Problem 5.12 (solution page 613)

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:

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.

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.