5.6 Eliminating Unneeded Memory References

The code for `combine3` accumulates the value being computed by the combining operation at the location designated by the pointer `dest`. This attribute can be seen by examining the assembly code generated for the inner loop of the compiled code. We show here the x86-64 code generated for data type `double` and with multiplication as the combining operation:

```
    Inner loop of combine3.  data_t = double, OP = *
    dest in %rbx, data+i in %rdx, data+length in %rax
1   .L17:                                   loop:
2     vmovsd  (%rbx), %xmm0                   Read product from dest
3     vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
4     vmovsd  %xmm0, (%rbx)                   Store product at dest
5     addq    $8, %rdx                        Increment data+i
6     cmpq    %rax, %rdx                      Compare to data+length
7     jne     .L17                            If !=, goto loop
```

We see in this loop code that the address corresponding to pointer `dest` is held in register `%rbx`. It has also transformed the code to maintain a pointer to the *i*th data element in register `%rdx`, shown in the annotations as `data+i`. This pointer is incremented by 8 on every iteration. The loop termination is detected by comparing this pointer to one stored in register `%rax`. We can see that the accumulated value is read from and written to memory on each iteration. 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. 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.

```
    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
```

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

```
1   /* Accumulate result in local variable */
2   void combine4(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long 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.

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine3`	549	Direct data access	7.17	9.02	9.02	11.03
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01

All of our times improve by factors ranging from 2.2× to 5.7×, with the integer addition case dropping to just 1.27 clock cycles per element.

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

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

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

Function	Initial	Before loop	`i` = 0	`i` = 1	`i` = 2	Final
`combine3`	[2, 3, 5]	[2, 3, 1]	[2, 3, 2]	[2, 3, 6]	[2, 3, 36]	[2, 3, 36]
`combine4`	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 30]

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

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

Practice Problem 5.4 (solution page 610)

When we use GCC to compile `combine3` with command-line option `-O2`, we get code with substantially better CPE performance than with `-O1`:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine3`	549	Compiled `-O1`	7.17	9.02	9.02	11.03
`combine3`	549	Compiled `-O2`	1.60	3.01	3.01	5.01
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01

We achieve performance comparable to that for `combine4`, except for the case of integer sum, but even it improves significantly. On examining the assembly code generated by the compiler, we find an interesting variant for the inner loop:

```
    Inner loop of combine3.  data_t = double, OP = *.  Compiled -O2
    dest in %rbx, data+i in %rdx, data+length in %rax
    Accumulated product in %xmm0
1   .L22:                                  loop:
2     vmulsd  (%rdx), %xmm0, %xmm0           Multiply product by data[i]
3     addq    $8, %rdx                       Increment data+i
4     cmpq    %rax, %rdx                     Compare to data+length
5     vmovsd  %xmm0, (%rbx)                  Store product at dest
6     jne     .L22                           If !=, goto loop
```

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

```
    Inner loop of combine3.  data_t = double, OP = *.  Compiled -O1
    dest in %rbx, data+i in %rdx, data+length in %rax
1   .L17:                                  loop:
2     vmovsd  (%rbx), %xmm0                  Read product from dest
3     vmulsd  (%rdx), %xmm0, %xmm0           Multiply product by data[i]
4     vmovsd  %xmm0, (%rbx)                  Store product at dest
```

```
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