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
1   .L5:                                          loop:
2     vmovss  -4(%rsi,%rax,4), %xmm0                Get p[i-1]
3     vaddss  (%rdi,%rax,4), %xmm0, %xmm0           Add a[i]
4     vmovss  %xmm0, (%rsi,%rax,4)                  Store at p[i]
5     addq    $1, %rax                              Increment i
6     cmpq    %rdx, %rax                            Compare i:cnt
7     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.

- Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.

Low-level optimizations. Structure code to take advantage of the hardware capabilities.

- Unroll loops to reduce overhead and to enable further optimizations.
- Find ways to increase instruction-level parallelism by techniques such as multiple accumulators and reassociation.
- Rewrite conditional operations in a functional style to enable compilation via conditional data transfers.

A final word of advice to the reader is to be vigilant to avoid introducing errors as you rewrite programs in the interest of efficiency. It is very easy to make mistakes when introducing new variables, changing loop bounds, and making the code more complex overall. One useful technique is to use checking code to test each version of a function as it is being optimized, to ensure no bugs are introduced during this process. Checking code applies a series of tests to the new versions of a function and makes sure they yield the same results as the original. The set of test cases must become more extensive with highly optimized code, since there are more cases to consider. For example, checking code that uses loop unrolling requires testing for many different loop bounds to make sure it handles all of the different possible numbers of single-step iterations required at the end.

5.14 Identifying and Eliminating Performance Bottlenecks

Up to this point, we have only considered optimizing small programs, where there is some clear place in the program that limits its performance and therefore should be the focus of our optimization efforts. When working with large programs, even knowing where to focus our optimization efforts can be difficult. In this section, we describe how to use *code profilers*, analysis tools that collect performance data about a program as it executes. We also discuss some general principles of code optimization, including the implications of Amdahl's law, introduced in Section 1.9.1.

5.14.1 Program Profiling

Program *profiling* involves running a version of a program in which instrumentation code has been incorporated to determine how much time the different parts of the program require. It can be very useful for identifying the parts of a program we should focus on in our optimization efforts. One strength of profiling is that it can be performed while running the actual program on realistic benchmark data.

Unix systems provide the profiling program GPROF. This program generates two forms of information. First, it determines how much CPU time was spent for each of the functions in the program. Second, it computes a count of how many times each function gets called, categorized by which function performs the call. Both forms of information can be quite useful. The timings give a sense of