A. The function performs $2n$ multiplications and n additions.

B. We can see that the performance limiting computation here is the repeated computation of the expression `xpwr = x * xpwr`. This requires a double-precision, floating-point multiplication (5 clock cycles), and the computation for one iteration cannot begin until the one for the previous iteration has completed. The updating of `result` only requires a floating-point addition (3 clock cycles) between successive iterations.

Solution to Problem 5.6 (page 508)

This problem demonstrates that minimizing the number of operations in a computation may not improve its performance.

A. The function performs n multiplications and n additions, half the number of multiplications as the original function `poly`.

B. We can see that the performance limiting computation here is the repeated computation of the expression `result = a[i] + x*result`. Starting from the value of `result` from the previous iteration, we must first multiply it by `x` (5 clock cycles) and then add it to `a[i]` (3 cycles) before we have the value for this iteration. Thus, each iteration imposes a minimum latency of 8 cycles, exactly our measured CPE.

C. Although each iteration in function `poly` requires two multiplications rather than one, only a single multiplication occurs along the critical path per iteration.

Solution to Problem 5.7 (page 510)

The following code directly follows the rules we have stated for unrolling a loop by some factor k:

```
1   void unroll5(vec_ptr v, data_t *dest)
2   {
3       long int i;
4       long int length = vec_length(v);
5       long int limit = length-4;
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       /* Combine 5 elements at a time */
10      for (i = 0; i < limit; i+=5) {
11          acc = acc OP data[i]   OP data[i+1];
12          acc = acc OP data[i+2] OP data[i+3];
13          acc = acc OP data[i+4];
14      }
15
16      /* Finish any remaining elements */
17      for (; i < length; i++) {
18          acc = acc OP data[i];
19      }
20      *dest = acc;
21  }
```


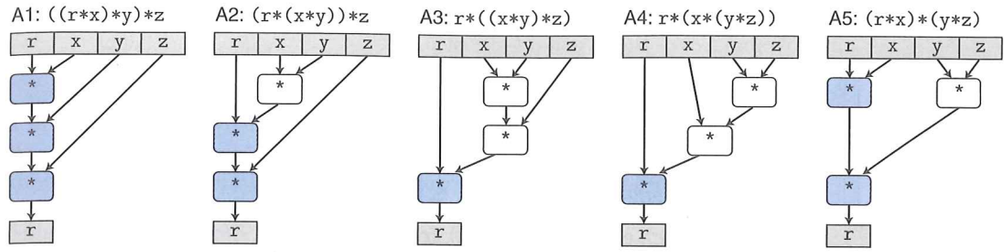

Figure 5.39 **Data dependencies among multiplication operations for cases in Problem 5.8.** The operations shown as blue boxes form the critical path for the iteration.

Solution to Problem 5.8 (page 523)

This problem demonstrates how small changes in a program can yield dramatic performance differences, especially on a machine with out-of-order execution. Figure 5.39 diagrams the three multiplication operations for a single iteration of the function. In this figure, the operations shown as blue boxes are along the critical path—they need to be computed in sequence to compute a new value for loop variable `r`. The operations shown as light boxes can be computed in parallel with the critical path operations. For a loop with c operations along the critical path, each iteration will require a minimum of $5c$ clock cycles and will compute the product for three elements, giving a lower bound on the CPE of $5c/3$. This implies lower bounds of 5.00 for A1, 3.33 for A2 and A5, and 1.67 for A3 and A4.

We ran these functions on an Intel Core i7, and indeed obtained CPEs of 5.00 for A1, and 1.67 for A3 and A4. For some reason, A2 and A5 achieved CPEs of just 3.67, indicating that the functions required 11 clock cycles per iteration rather than the predicted 10.

Solution to Problem 5.9 (page 530)

This is another demonstration that a slight change in coding style can make it much easier for the compiler to detect opportunities to use conditional moves:

```
while (i1 < n && i2 < n) {
    int v1 = src1[i1];
    int v2 = src2[i2];
    int take1 = v1 < v2;
    dest[id++] = take1 ? v1 : v2;
    i1 += take1;
    i2 += (1-take1);
}
```

We measured a CPE of around 11.50 for this version of the code, a significant improvement over the original CPE of 17.50.

Solution to Problem 5.10 (page 538)

This problem requires you to analyze the potential load-store interactions in a program.

A. It will set each element `a[i]` to $i + 1$, for $0 \le i \le 998$.

B. It will set each element `a[i]` to 0, for $1 \le i \le 999$.

C. In the second case, the load of one iteration depends on the result of the store from the previous iteration. Thus, there is a write/read dependency between successive iterations. It is interesting to note that the CPE of 5.00 is 1 less than we measured for Example B of function `write_read`. This is due to the fact that `write_read` increments the value before storing it, requiring one clock cycle.

D. It will give a CPE of 2.00, the same as for Example A, since there are no dependencies between stores and subsequent loads.

Solution to Problem 5.11 (page 538)

We can see that this function has a write/read dependency between successive iterations—the destination value `p[i]` on one iteration matches the source value `p[i-1]` on the next.

Solution to Problem 5.12 (page 539)

Here is a revised version of the function:

```
1  void psum1a(float a[], float p[], long int n)
2  {
3      long int i;
4      /* last_val holds p[i-1]; val holds p[i] */
5      float last_val, val;
6      last_val = p[0] = a[0];
7      for (i = 1; i < n; i++) {
8          val  = last_val + a[i];
9          p[i] = val;
10         last_val = val;
11     }
12 }
```

We introduce a local variable `last_val`. At the start of iteration `i`, it holds the value of `p[i-1]`. We then compute `val` to be the value of `p[i]` and to be the new value for `last_val`.

This version compiles to the following assembly code:

```
   psum1a.  a in %rdi, p in %rsi, i in %rax, cnt in %rdx, last_val in %xmm0
1   .L18:                                  loop:
2     addss   (%rdi,%rax,4), %xmm0          last_val = val = last_val + a[i]
3     movss   %xmm0, (%rsi,%rax,4)          Store val in p[i]
4     addq    $1, %rax                      Increment i
5     cmpq    %rax, %rdx                    Compare cnt:i
6     jg      .L18                          If >, goto loop
```


This code holds `last_val` in `%xmm0`, avoiding the need to read `p[i-1]` from memory, and thus eliminating the write/read dependency seen in `psum1`.

Solution to Problem 5.13 (page 546)

This problem illustrates that Amdahl's law applies to more than just computer systems.

A. In terms of Equation 5.4, we have $\alpha = 0.6$ and $k = 1.5$. More directly, traveling the 1500 kilometers through Montana will require 10 hours, and the rest of the trip also requires 10 hours. This will give a speedup of $25/(10 + 10) = 1.25$.

B. In terms of Equation 5.4, we have $\alpha = 0.6$, and we require $S = 5/3$, from which we can solve for k. More directly, to speed up the trip by 5/3, we must decrease the overall time to 15 hours. The parts outside of Montana will still require 10 hours, so we must drive through Montana in 5 hours. This requires traveling at 300 km/hr, which is pretty fast for a truck!

Solution to Problem 5.14 (page 546)

Amdahl's law is best understood by working through some examples. This one requires you to look at Equation 5.4 from an unusual perspective.

This problem is a simple application of the equation. You are given $S = 2$ and $\alpha = .8$, and you must then solve for k:

$$2 = \frac{1}{(1 - 0.8) + 0.8/k}$$

$$0.4 + 1.6/k = 1.0$$

$$k = 2.67$$