Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 5: Optimizing Program Performance April 25, 2024

Practice Problems

Exercise 5.1. The following problem illustrates the way memory aliasing can cause unexpected program behavior. Consider the following procedure to swap to values:

```
/* Swap value x at xp with value y at yp */
void swap(long *xp, long *yp)
{
    *xp = *xp + *yp;  /* x + y */
    *yp = *xp - *yp;  /* x+y-y = x */
    *xp = *xp - *yp;  /* x+y-x = y */
}
```

If this procedure is called with `xp` equal to `yp`, what effect will it have?

Solution: If `xp` equals `yp`, meaning that the pointers hold the same memory address, then the variables are aliased. As a result, all subsequent assignments set both `*xp` and `*yp`. The first expression sets `*xp` (and hence the aliased `*yp`) to twice the original value of `*x`. Then, the next expression evaluates to 0, so `*yp` and `*xp` are set 0. The final expression sets `*xp` (and hence `*yp`) to `0 - 0`, or just `0`.

Therefore, instead of swapping values, both values are set to `0`.

Exercise 5.2. Later in this chapter we will start with a single function and generate many different variants that preserve the function's behavior, but with difference performance characteristics. For three of these variants, we found that the run times (in clock cycles) can be approximated by the following functions:

- Version 1: $60 + 35n$
- Version 2: $136 + 4n$
- Version 3: $157 + 1.25n$

For what values of n would each version be the fastest of the three? Remember that n will always be an integer.

Solution: When $n = 0$, Version 1 has the smallest value: 60. That is, it requires the least cycles per elements. Because it has the greatest slope, it will eventually surpass both of the other versions in terms of required cycles per element. Version 1 will intersect Version 2 when $60 + 35n = 136 + 4n$, or $31n = 76$, making n about 2.45. Since n is an integer, this means we require n to be at least 3. Similarly, Version 1 and Version 3 intersect when $60 + 35n = 157 + 1.25n$. This means $33.75n = 97$, so n is about 2.87, but once again n must be an integer so we require it to be 3. At this point, either Version 2 or Version 3 is the fastest. These versions intersect when $136 + 4n = 157 + 1.25n$, so $2.75n = 21$, meaning n is

about 7.6. Version 2 hs a larger slope, so eventually its slope will overcome that of Version 3,; this will happen when $n = 8$. However, this means that when n is between 3 and 7 (inclusive), Version 2 will have less cycles per element.

Therefore, when $n < 3$, Version 1 is the fastest, followed by Version 2 when $3 \leq n < 7$, and lastly, Version 3 is the fastest when $n \geq 8$, requiring 1.25 cycles per element.

Exercise 5.3. Consider the following functions:

```
long min(long x, long y) { return x < y ? x : y; }
long max(long x, long y) { return x < y ? y : x; }
void incr(long *xp, long v) { *xp += v; }
long square(long x) { return x*x; }
```

The following three code fragments call these functions:

(a)

```
for (i = min(x, y); i < max(x, y); incr(&i, 1)
    t += square(i);
```

(b)

```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

(c)

```
long low = min(x, y);
long high = max(x, y);
for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

Assume `x` equals 10 and `y` equals 100. Fill int he following table indicating the number of times each of the four functions is called in code fragments A-C.

Code	`min`	`max`	`incr`	`square`
A	_____	_____	_____	_____
B	_____	_____	_____	_____
C	_____	_____	_____	_____

Solution:

Code	`min`	`max`	`incr`	`square`
A	1	91	90	90
B	91	1	90	90
C	1	1	90	90

Exercise 5.4. 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`	513	Compiled `-O1`	7.17	9.02	9.02	11.03
`combine3`	513	Compiled `-O2`	1.60	3.01	3.01	5.01
`combine4`	513	Accumulate in temporary	1.27	3.01	3.01	5.01

We achieve performance comparable to that of `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 of 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
.L22:                           # loop:
    vmulsd (%rdx), %xmm0, %xmm0 #   Multiply product by data[i]
    addq   $8, %rdx             #   Increment data + i
    cmpq   %rax, %rdx           #   Compare to data+length
    vmovsd %xmm0, (%rbx)        #   Store product at dest
    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
.L17:                           # loop:
    vmovsd (%rbx), %xmm0        #   Read product from dest
    vmulsd (%rdx), %xmm0, %xmm0 #   Multiply product by data[i]
    vmovsd %xmm0, (%rbx)        #   Store product at dest
    addq   $8, %rdx             #   Increment data + i
    cmpq   %rax, %rdx           #   Compare to data+length
    jne    .L22                 #   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.

Solution:

(a) In the version of `compare3` compiled with `-O2`, the `%xmm0` register holds the value accumulated so far. Since this value is used in the next iteration, there is no need to read memory to obtain it; it can be directly used from `%xmm0`. This is in contrast with with the version of `compare3` compiled with `-O1`, where rather than reading the accumulated value from `%xmm0`, it is read from memory before operating on it.

(b) Yes, the more optimized version faithfully implements `combine3`. In both cases, the computed value is written to the destination every iteration. Therefore, the value read from the accumulated register `%xmm0` in the version compiled with `-O2` will always match the value read via the memory reference `(%rbx)` in version compiled with `-O1`.

(c) The optimized version preserves the desired behavior by updating the destination with the current value in `dest` in every iteration, rather than doing so once at the end after the loop ends.

Exercise 5.5. 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 $a_0, a_1, \ldots, a_n$. For a value x, we evaluate the polynomial by computing

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

This evaluation can be implemented by the following function, having as arguments an array of coefficients `a`, a value `x`, and the polynomial of degree `degree` (the value n in the polynomial expression above). In this function, we compute both the successive terms of the equation and the successive powers of x within a single loop:

```
double poly(double a[], double x, long degree)
{
    long i;
    double result = a[0];
    double xpwr = x;  /* Equals x^i at start of loop */
    for (i = 1; i <= degree; i++) {
        result += a[i] * xpwr; /* Line 7 */
        xpwr = x * xpwr;      /* Line 8 */
    }
    return result;
}
```

(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 due to the operations implementing lines 7-8 of the function.

Solution:

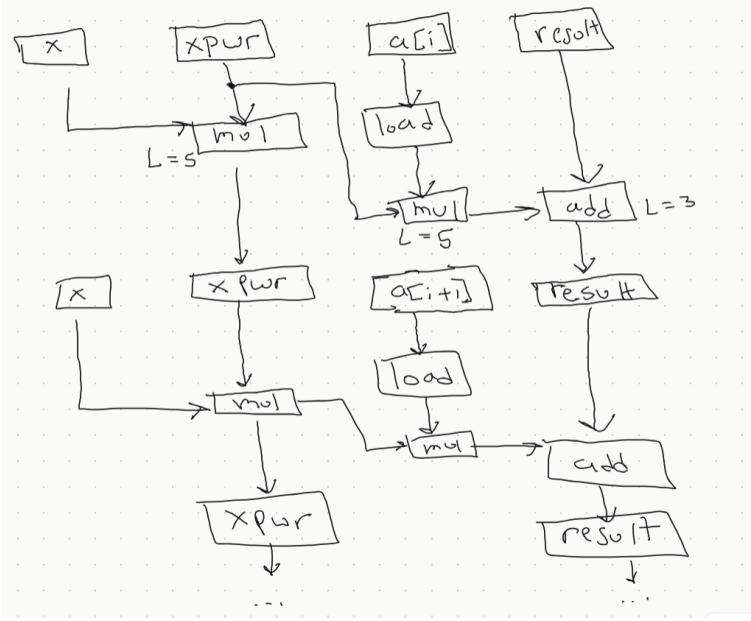

Figure 1: Exercise 05-05: Data flow graph representing dependencies during i-th iteration

(a) There are n additions due to the statement `i++`, and n additions due to the `result += ` statement, for a total of $2n$ additions. There are $2n$ multiplications, where n come from the first statement in the loop which multiplies a coefficient by a power of x, and n from the second coming from computing the next power of x.

(b) On line 7, we see that `result` is used as both a source and destination, making its associated floating point register a *loop* register. The `a[i]` and `x` expressions are associated with a *read-only* register, since they are only used as source values and not updated Finally, the `xpwr` expression is used as a source and destination value, making its associated register at `loop` register. See Figure 1.

At first, it may seem like the latency is 8 cycles because the addition on line 7 cannot start until the product `a[i] * xpwr` is computed. Let the moment before the first iteration correspond to cycle 0. Since the capacity of the floating point multiplication is 2 and the latency is 5, and the two products in each iteration are independent, the two products `a[0] * pwr` and `x * xpwr` occur simultaneously.

On cycle 5, the product `a[0] * xpwr` has completed, so the addition portion of `result += a[0] * xpwr` may begin. Similarly, the floating-point multiplication operations `a[1] * xpwr` and `x * xpwr` can begin also. Since the latency of floating point addition is 3 cycles, the addition `result += a[0] * xpwr` finishes by cycle 8. However, the multiplications are still happening. As a result, the addition in `result += a[1] * xpwr` cannot begin. By cycle 10, the two multiplications from the second iteration have ended, and at this point the addition `result += a[1] * xpwr` can take place, ending at cycle 13.

Altogether, the `xpwr` variable updates every 5 cycles, while the `result` variable also updates every 5 cycles. The difference is that `result` lags 3 cycles behind. For example,

the `xpwr` variable is updated at cycles 5, 10, 15, and so on. Meanwhile, the `result` variable is updated at cycles 8, 13, 18, and so on, After n iterations, the last value of `xpwr` will have computed after $5n$ cycles, whereas the last value of `result` will be computed after $5n + 3$ cycles.

Exercise 5.6. 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 (1787-1837). The idea is to repeatedly factor out powers of x to get the following evaluation:

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

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

```
/* Apply Horner's method */
double polyh(double a[], double x, long degree)
{
    long t;
    double result = a[degree];
    for (i = degree-1; i>=0 ; i--)
        result = a[i] + x * result; /* Line 7 */
    return result;
}
```

(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 this function.

(c) Explain how the function shown in Practice Problem 5.5 can run faster, even though it requires more operations.

Solution:

(a) Considering `i--` to be an addition of `-1`, there are n additions attributed to it, and n additions attributed to the loop statement. There are n multiplications inside the loop. Therefore, there's a total of $2n$ additions and n multiplications.

(b) Figure 2 shows the data graph representing the dependencies in the i-th iteration. The `result` variable goes through two operations in series; multiplication by `x` and then it is added to `a[i]`. The multiplication takes 5 cycles while the addition takes 3 cycles (on the reference machine), so the latency is at least 8 cycles, consistent with the CPE.

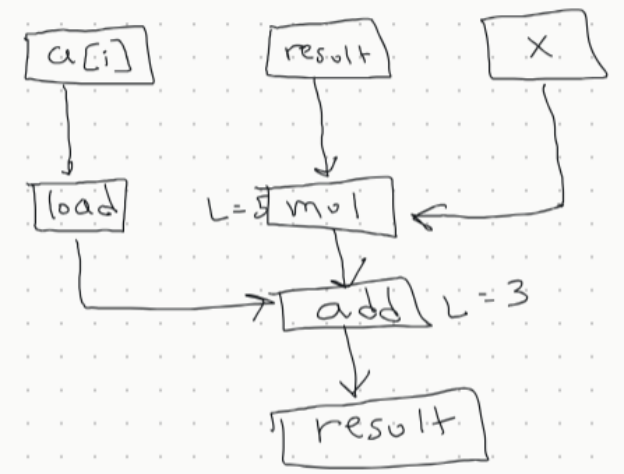

Figure 2: Exercise 05-06: Data-flow graph representing dependencies during i-th iteration

(c) The critical path in Practice Problem 5.5 involves only multiplication, which takes 5 cycles. Because the capacity is 2, both floating-point multiplications occur simultaneously, taking up 5 cycles. The addition depends on the result of the first of these two products, and thus it can occur every 5 cycles. The critical path is therefore the chain of `xpwr` computations.

In contrast, Horner's method introduces a data dependency, whereby as indicated by Figure 2. Now both multiplication and addition are on the critical path of `result`, leading to the higher CPE.

Exercise 5.7. Modify the code for `combine5` to unroll the loop by a factor of $k = 5$.

Solution: For 5×1 loop unrolling, we increment the iteration index by 5 each time. Also, we operate on 5 operands. To ensure we do not overrun the array bounds, we set the limit to $length - 5 + 1$:

```
/* 5 x 1 loop unrolling */
void combine5(vec_ptr, data_t *dst)
{
    long i;
    long length = vec_length(v);
    long limit = length - 5 + 1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 5) {
        acc = ((((acc OP data[i])
                            OP data[i+1])
                            OP data[i+2])
                            OP data[i+3])
                            OP data[i+4];
```

```
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Exercise 5.8. Consider the following function for computing the product of an array of n double-precision numbers. We have unrolled the loop by a factor of 3.

```
double aprod(double a[], long n)
{
    long i;
    double x, y, z;
    double r = 1;
    for (i = 0; i < n-2; i += 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; /* Product computation */
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

For the line labeled "`Product computation`," we can use parentheses to create five different associations of the computation as follows:

```
r = ((r * x) * y) * z; /* A1 */
r = (r * (x * y)) * z; /* A2 */
r = r * ((x * y) * z); /* A3 */
r = r * (x * (y * z)); /* A4 */
r = (r * x) * (y * z); /* A5 */
```

Assume we run these functions on a machine where floating-point multiplication has a latency of 5 clock cycles. Determine the lower bound on the CPE set by the data dependencies of the multiplication. (`Hint`: It helps to draw a data-flow representation of how `r` is computed on every iteration.)

Solution: For `A1`, see the data flow diagram in Figure 3. There are 3 multiplications for each iteration on the critical path, and around $n/3$ iterations, so overall there are n multiplications in the critical path. Since the latency of floating-point multiplication is 5 clock cycles, this leads to a CPE of around 5.0.

For `A2`, see the data flow diagram in Figure 4. The first multiplication on the critical path to compute `r` cannot start until the product `x * y` has been computed. The effect of this is that there is a lag of 5 clock cycles, but ultimately, there are 2 multiplications on the

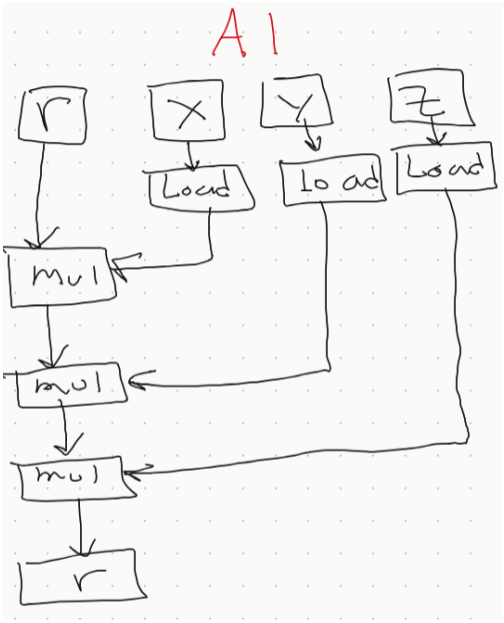

Figure 3: Exercise 05-08: Association A1

critical path of for each iteration. With around $n/3$ iterations, this means there are around $2n/3$ multiplications altogether. Since each multiplication has a latency of 5 clock cycles, this leads to a CPE of around $5 \cdot \frac{2}{3} \approx 3.33$.

For `A3`, see the data flow diagram in Figure 5. In cycle 0, the product `x * y`, which is `a[0] * a[1]` is computed, while the other ones are put on hold since the depend on this result. In cycle 5, the product `(x * y) * z` is computed with the result of `x * y` that has already been computed, but the product involving `r` is still on hold to wait for this result. However, the product `x * y` or equivalently `a[3] * a[4]` in the next iteration starts in cycle 5 as well. In cycle 10, `r` begins to compute, and `(x * y) * z` also begins to compute for the next iteration, because `x * y` has already been computed. Moreover, `a[6] * a[7]` from the next iteration also begins in cycle 10. In cycle 15, `(a[3] * a[4]) * a[5]` from the previous iteration has already computed, so the next value of `r` can compute, while `(a[6] * a[7]) * a[8]` and and `a[10] * a[11]` compute. In cycle 20, the next value of `r` can compute as `r * ((a[6] * a[7]) * a[8])`, while `(a[10] * a[11]) * a[12]` and `a[13] + a[14]` compute. Altogether, `r` begins to compute every 5 cycles, lagging behind the other products. Since there are $n/3$ iterations, this implies a CPE of around $5 \cdot (1/3) \approx 1.67$.

For `A4`, see the data flow diagram in Figure 6. The CPE is around 5.0 just like in `A3`.

For `A5`, see the data flow diagram in Figure 7. This version has 2 multiplications on the critical path, so it has a CPE of around 3.33 just like in `A2`.

Exercise 5.9. The traditional implementation of the merge step of mergesort requires three loops:

```
void merge(long src1[], long src2[], long dest[], long n) {
    long i1 = 0;
    long i2 = 0;
```

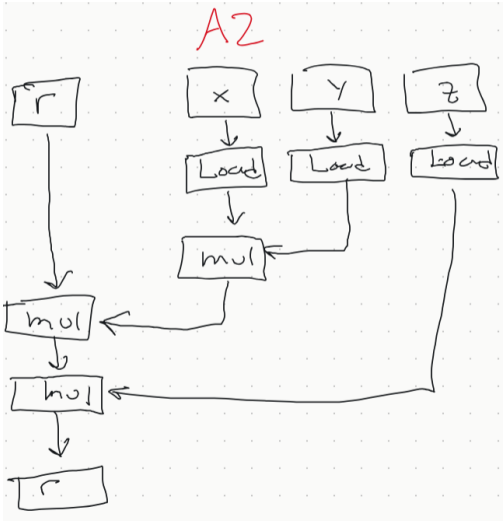

Figure 4: Exercise 05-08: Association A2

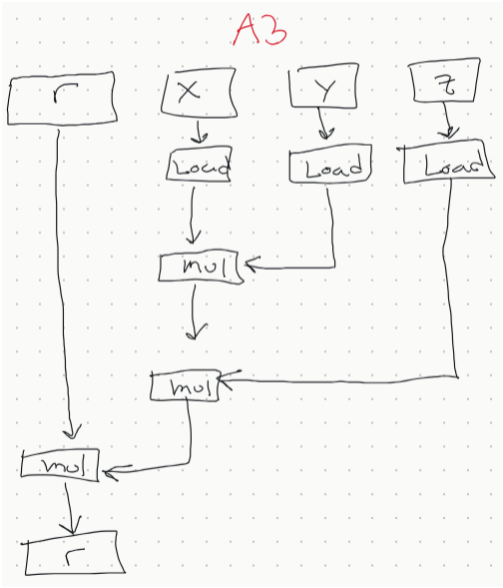

Figure 5: Exercise 05-08: Association A3

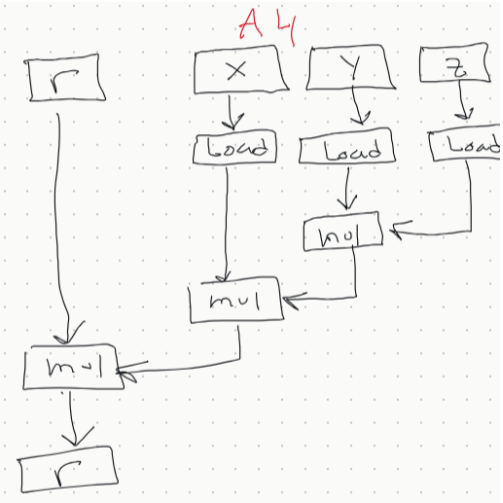

Figure 6: Exercise 05-08: Association A4

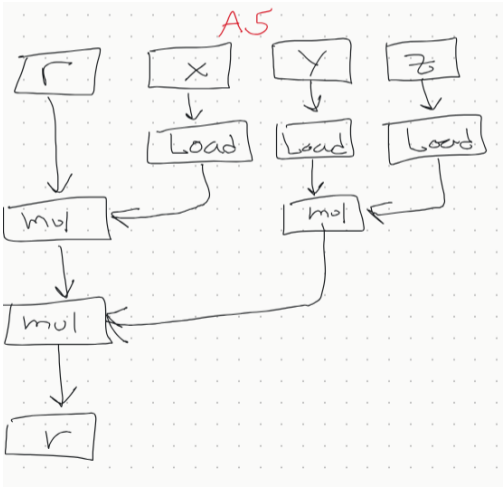

Figure 7: Exercise 05-08: Association A5

```
    long id = 0;
    while (i1 < n && i2 < n) {
        if (src1[i1] < src2[i2]) /* line 6 */
            dest[id++] = src1[i1++];
        else
            dest[id++] = src1[i2++];
    }
    while (i1 < n)
        dest[id++] = src1[i1++];
    while (i2 < n)
        dest[id++] = src2[i2++];
}
```

The branches caused by comparing variables `i1` and `i2` to `n` have good prediction performance — the only mispredictions occurs 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.

Solution: The book mentions that a functional style, where conditional operations are used to compute values and then update program state with these values, are more amenable to translation into conditional data transfers than an imperative style, where conditionals are used to selectively update program state. This means we need to eliminate the `if-else` statement.

The solution provided by the authors, presented below, makes use of the fact that the result of the assignment and index updates depend on the result of `a < b`. In C, the result is 1 when the comparison is true, so the 1 can be used to increase `i1`; similarly, `1 - aLTb` will be 0 in this case, which can be used to update `i2`. Hence, `aLTb` is used to assign the correct value to `dest` and the values of `i1`, `i2`, and `id` are computed concurrently. Together with the fact that `a` and `b` are fetched concurrently, the code becomes amenable to implementation with a conditional move.

```
void merge(long src1[], long src2[], long dest[], long n) {
    long i1 = 0;
    long i2 = 0;
    long id = 0;
    while (i1 < n && i2 < n) {
        long a = src1[i1];
        long b = src2[i2];
        int aLTb = a < b;
        dest[id++] = aLTb ? a : b;
        i1 += aLTb;
        i2 += (1-aLTb);
    }
    while (i1 < n)
```

```
        dest[id++] = src1[i1++];
    while (i2 < n)
        dest[id++] = src2[i2++];
}
```

Exercise 5.10. 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];
}
```

Suppose `a` is an array of length 1000 initialized so that each element of `a[i]` equals i.

(a) What would be the effect of calling `copy_array(a+1, a, 999)`?

(b) What would be the effect of calling `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)`

Solution:

(a) Initially, the array has values 0 through 999, inclusive. Afterwards, it will have values 1 through 999 at positions 0 through 998, respectively. The value at position 999, namely `a[999]`, will remain unchanged, with a value of 999.

(b) The computation sets all elements to the value of `a[0]`, which is 0.

(c) In part A, the load operation and store operation refer to distinct addresses, so they can compute independently. In part B, the load operation depends on the pending store operation from the previous iteration, introducing a data dependency that becomes the critical path. Since the reference machine has a 4 cycle access time, this must complete before it can be accessed for the next iteration, leading to the 5.0 CPE.

(d) I would expect a CPE of 1.0 because even though the store and read address are the same, there is no pending store. In other words, there is no pending right when the address is initially computed; the write happens afterwards. Therefore, there is no dependency that would lead to the 4.0 CPE penalty described above.

Exercise 5.11. 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 `combine4` (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.

Solution: The following is the code for `p_sum1`:

```
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long n)
{
    long i;
    p[0] = a[0];
    for (i= 1; i < n; i++)
        p[i] = p[i-1] + a[i];
}
```

There is a data dependency due to the fact that `p[i]` is written to in the i-th iteration, and read from in the $(i+1)$-th iteration. Only then can the floating-point addition with `a[i]` take place. Therefore the critical path contains the store operation, the load operation, and the floating-point addition. The floating-point addition incurs 3 clock cycles on the CPE. The dependency between the store and load operations incurs a penalty of about 6 clock cycles on the CPE, explaining the 9.0 CPE.

Exercise 5.12. Rewrite the code for `p_sum1` (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 the floating-point addition.

Solution: To eliminate the dependency, I have introduced an accumulator variable `acc`, playing the role of `p[i-1]` in the original implementation. Its value is computed by adding `a[i]`, giving the new value of `p[i]`. This will be considered the old value in the next iteration, which instead of being read from `p[i-1]` will already be present in `acc`.

```
/* Compute prefix sum of vector a */
void psum1_improved(float a[], float p[], long n) {
    long i;
    float acc = a[0];
    p[0] = acc;
    for (i=1; i < n; i++) {
        acc += a[i];
        p[i] = acc;
    }
}
```

Exercise 5.13. Suppose we wish to write a procedure that computes the inner product of two vectors `u` and `v`, An abstract version of the function has a CPE of 14–18 with x86-64 for different types of integer and floating-point data. By doing the same sort of transformations we did to transform the abstract program `combine1` into the more efficient `combine4`, we get the following code:

```
/* Inner product. Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;

    for (i = 0; i < length; i++) {
        sum = sum + udata[i] * vdata[i];
    }
    *dest = sum;
}
```

Our measurements show that this function has CPEs of 1.50 for integer data and 3.00 for floating-point data. For data type `double`, the x86-64 assembly code for the inner loop is as follows:

```
# Inner loop of inner4, data_t = double, OP = *
udata in %rbp, vdata in %rax, sum in %xmm0
.L15:                                   loop:
    xmovsd 0(%rbp,%rcx,8), %xmm1            Get udata[i]
    vmulsd (%rax,%rcx,8), %xmm1, %xmm1      Multiply by vdata[i]
    vaddsd %xmm1, %xmm0, %xmm0              Add to sum
    addq   $1, %rcx                         Increment i
    cmpq   %rbx, %rcx                       Compare i:limit
    jne    .L15                             If !=, goto loop
```

Assume that the function units have the characteristics listed in Figure 5.12. Namely:

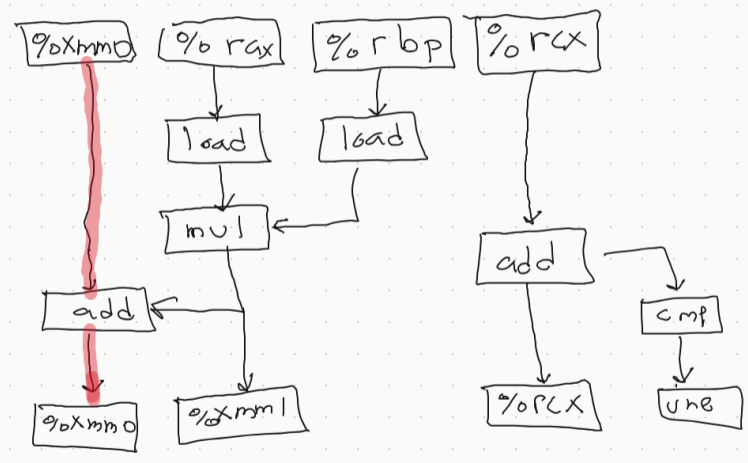

Figure 8: Exercise 5-13a: Data-dependency flow diagram

	Integer			Floating point		
Operaton	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

(a) Diagram how this instruction sequence would be decoded into operations and show how the data dependencies between them would create a critical path of operations, in the style of Figures 5.13 and 5.14.

(b) For data type `double`, what lower bound on the CPE is determined by the critical path?

(c) Assuming similar instruction sequences for the integer code as well, what lower bound on the CPE is determined by the critical path for integer data?

(d) Explain how the floating-point versions can have CPEs of 3.00, even though the multiplication operation requires 5 clock cycles.

Solution:

(a) See Figure 8.

(b) The critical involves the operations updating `%xmm0`, the register of the accumulation variable `sum`, via the `add` operation. The multiplication is not involved in a data-dependency chain with loop registers, so it can be pipelined. That means that although any one multiplzugh the multiplication for the first iteration has not completed on cycle 1, the multiplication for the second iteration can still begin on that cycle. Therefore, at cycle 3, the multiplication for the first iteration is available, at cycle 4 the multiplication for the second iteration is available, and so on.

Since the loop register `%xmm0` depends on the floating point addition, the fact that it has a latency of 3 on the reference machine determines a lower bound of 3.0 for the CPE.

(c) A similar argument applies as before, so both loop registers `%rcx` and the register holding `%sum` constitute critical paths. Since addition has a latency of 1, the lower bound for the CPE is 1.0 for integer data. The implementation's CPE is 1.50, meaning that the bottleneck is elsewhere, such as the overhead of comparison operations and loads.

(d) As explained in part (b), pipelining and the fact that the multiplication is not in the critical path implies that the results of the multiplications will be available on-demand after the first iteration. For example, on cycle 0 the product `udata[0] * vdata[0]` is issued, and its final value is not determined until 5 cycles later, delaying the first addition. However, the other products do not depend on this result, so aon cycle 1 the computation `udata[1] * vdata[1]` begin (finished by cycle 6), on cycle 2 the computation `udata[2] * vdata[2]` begins (finished by cycle 7), and so on. This means that the floating-point addition becomes the bottleneck, since it involves a loop register.

Exercise 5.14. Write a version of the inner product procedure described in Problem 5.13 that uses 6×1 loop unrolling. For x86-64, our measurements of the unrolled version gives a CPE of 1.07 for integer data but still 3.01 for both floating-point data.

(a) Explain why any (scalar) version of an inner product procedure running on an Intel Core i7 Haswell processor cannot achieve a CPE less than 1.00.

(b) Explain why the performance for floating-point data did not improve with loop unrolling.

Solution: My version with 6×1 loop unrolling is given below:

```
/* Inner product. Accumulate in temporary, 6 by 1 loop unrolling */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    long limit = length - 5;     /* length - (k - 1), where k = 6 */
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;

    for (i = 0; i < limit; i+= 6) { /* use limit instead of length, and increment
        by 6 */
        sum = sum + (udata[i] * vdata[i]) + (udata[i+1] * vdata[i+1]);
        sum = sum + (udata[i+2] * vdata[i+2]) + (udata[i+3] * vdata[i+3]);
        sum = sum + (udata[i+4] * vdata[i+4]) + (udata[i+5] * vdata[i+5]);
    }

    /* Finish any remaining elements */
    for (; i < length ; i++) {
        sum = sum + (udata[i] * vdata[i]);
```

```
    }
    *dest = sum;
}
```

(a) Because of the dependency chain formed by `add` involving the register holding the `sum` variable, we will continue to have a chain of n additions in the critical path, where n is the length of the vectors. This means that the CPE has the latency of the operation in question as its lower bound. The smallest such latency is that of integer addition, so the theoretical lower bound is 1.

(b) Loop unrolling with $k = 6$ means that there's about $1/6$ of the iterations as we would otherwise have without unrolling, but each iteration has $k = 6$ additions in sequence along each iteration of the critical path. Therefore, the overall number of `add` operations remains as n, the number of elements in the vectors.

Exercise 5.15. Write a version of the inner product procedure described in Problem 5.13 that uses 6×6 loop unrolling. Our measurements for this function with x86-64 give a CPE of 1.06 for integer data and 1.01 for floating-point data. What factor limits the performance to a CPE of 1.00?

Solution: My version of the inner product procedure with 6×6 loop unrolling is below:

```
/* Inner product. Accumulate in temporary, 6 by 1 loop unrolling */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    long limit = length - 5;     /* length - (k - 1), where k = 6 */
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum0 = (data_t) 0;    /* Multiple accumulators */
    data_t sum1 = (data_t) 0;
    data_t sum2 = (data_t) 0;
    data_t sum3 = (data_t) 0;
    data_t sum4 = (data_t) 0;
    data_t sum5 = (data_t) 0;

    for (i = 0; i < limit; i+= 6) { /* use limit instead of length, and increment
        by 6 */
        sum0 = sum0 + (udata[i] * vdata[i]);
        sum1 = sum1 + (udata[i+1] * vdata[i+1]);
        sum2 = sum2 + (udata[i+2] * vdata[i+2]);
        sum3 = sum3 + (udata[i+3] * vdata[i+3]);
        sum4 = sum4 + (udata[i+4] * vdata[i+4]);
        sum5 = sum5 + (udata[i+5] * vdata[i+5]);
    }
```

```
    /* Finish any remaining elements */
    for (; i < length ; i++) {
        sum0 = sum0 + (udata[i] * vdata[i]);
    }
    *dest = sum0 + sum1 + sum2 + sum3 + sum4 + sum5;
}
```

In Practice Problem 5.14, the use of loop unrolling improved performance but since we only had one accumulator. As a result it could not leverage instruction-level parallelism, bounding the CPE by the latency. There, it was limited to one addition operation every L cycles, where L is the latency of the addition operation on the critical path; see Section 5.9. In this problem, the use of 6×6 loop unrolling means we use 6 accumulation variables (here `sum0` through `sum5`). Therefore, the processor does not need to delay one sum until the previous one has completed.

For example, consider the case of floating-point addition with a `double` where the latency is 3 cycles. The processor no longer needs to wait 3 cycles to issue a new addition operation. Instead, since the issue time is $I = 1$ and capacity is $C = 1$ (see the table on Practice Problem 5.13), the throughput bound is $C/I = 1$ (see page 524 on Section 5.7.2). To achieve this throughput, we needed the unrolling factor to be $k \geq C \cdot L$, where L is the latency. For floating-point addition, this means we need $k \geq 1 \cdot 3 = 3$, so $k = 6$ is sufficient. This explains the improvement for the floating-point addition. However, the implementation cannot do better than this floating-point addition throughput bound of 1.

In the case of integer addition, the throughput bound is around 0.50 because even though it has a capacity time and issue time of 1, the CPU in question has only 2 load units. However, the limiting factor becomes integer multiplication, which a minimum throughput of 1.

Exercise 5.16. Write a version of the inner product procedure described in Problem 5.13 that uses $6 \times 1a$ loop unrolling to enable greater parallelism. Our measurements for this function gives a CPE of 1.10 for integer data and 1.05 for floating-point data.

Solution: My implementation of the inner product procedure using $6 \times 1a$ loop unrolling is below:

```
/* Inner product. Accumulate in temporary, 6 by 1a loop unrolling (association)
   */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    long limit = length - 5;     /* length - (k - 1), where k = 6 */
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;

    for (i = 0; i < limit; i+= 6) { /* use limit instead of length, and increment
       by 6 */
```

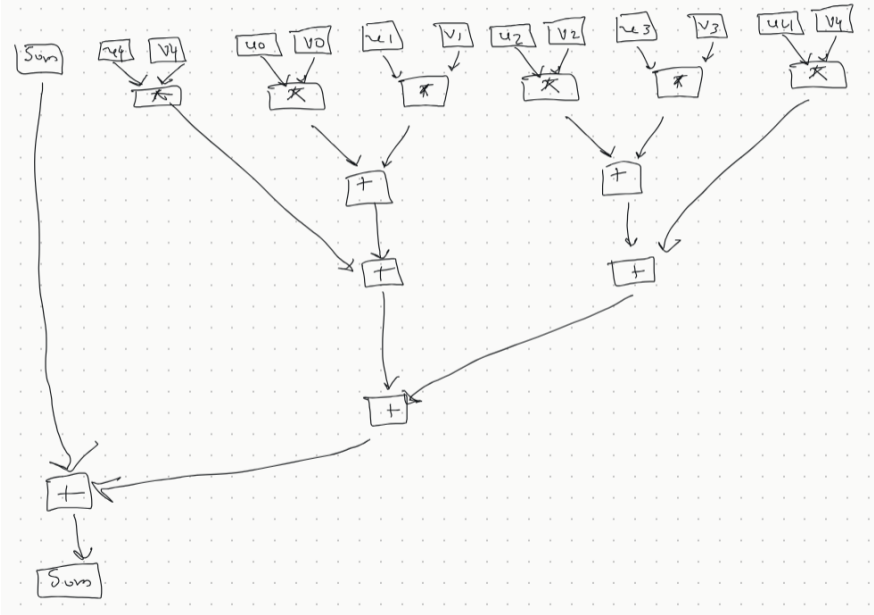

Figure 9: Exercise 5-16: Data dependency flow graph for inner product using $6 \times 1a$ loop unrolling

```
        data_t u0 = udata[i], v0 = udata[i];
        data_t u1 = udata[i+1], v1 = udata[i+1];
        data_t u2 = udata[i+2], v2 = udata[i+2];
        data_t u3 = udata[i+3], v3 = udata[i+3];
        data_t u4 = udata[i+4], v4 = udata[i+4];
        data_t u5 = udata[i+5], v5 = udata[i+5];
        data_t x0 = u0 * v0;
        data_t x1 = u1 * v1;
        data_t x2 = u2 * v2;
        data_t x3 = u3 * v3;
        data_t x4 = u4 * v4;
        data_t x5 = u5 * v5;
        sum = sum + ((x4 + (x0 + x1)) + (x5 + (x2 + x3)));
    }

    /* Finish any remaining elements */
    for (; i < length ; i++) {
        sum = sum + (udata[i] * vdata[i]);
    }
    *dest = sum;
}
```

It uses reassociation to change the dependency chains in the critical path to achieve a higher CPE. See Figure 9.