Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 5: Optimizing Program Performance April 19, 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:

```c
/* 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. The first expression sets `*xp` to `2 * x`, twice the original value of `x`. It also inadvertently changes `*yp` to have value `2 * x`. Then, the next expression evaluates to 0, so `*yp` and hence `*xp` is 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. Version 1 will intersect Version 2 whe $60 + 35n = 136 + 4n$, 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)