Lab Week #8
CIS 314

November 14, 2017

1 C Speed Optimizations

The book does a great job with optimizations, but there are more that exist (and many more than are mentioned here!). Nevertheless, here are the major categories of optimizations that will frequently be needed.

1.1 Perspective

Frequently beginning programmers will make the mistake of creating a 'logical detour' (i.e. some statement which could have been avoided entirely). Consider the following example:

```
if( x!=0 ){
  x=0;
}
//The above is functionally equivalent to the following
x=0;
```

More commonly, 100 level students will attempt to flip a boolean value with the following code:

```
//We're in some language with bools...
if( x == true ){
  x = false;
}
else{
  x = true;
}
//The above is functionally equivalent to the following
x=!x;
```

1.2 Inlining

Recall the following example, shamelessly stolen from the book:

```
long counter = 0;

inline long f(){
  return counter++;
}

long func1(){
  return f() + f() + f() + f();
}
```

Indeed, the compiler will not remove the function calls, even with the *-O1* flag. We can, however, force it (well... encourage it) to inline the function via several methods. The first would be to simply issue the *-finline-functions* flag, but this doesn't work in Clang, and hence will not work with the aliased GCC compiler that comes with macOS. The next technique is to prepend the keyword *inline* to each function that we want to be inlined, but this still doesn't convince the compiler to inline our function in this case. When all else fails, a macro containing the function body will do the trick:

```
#define f() (counter++)
long counter = 0;

long func1(){
  return f() + f() + f() + f();
}
```

There is a caveat here – if you have a very large function, the act of inlining it may force your code to reside in some cache that is farther from the CPU... you may end up with slower code!

1.3 Unroll loops

Imagine the following code block to sum the elements of an array of length 256:

```
int sum(int a[]){

  int i, sum = 0;
  for(i=0;i<256;i++){
    sum += a[i];
  }
  return sum;
}
```

By using the -funroll-loops, you can tell your compiler to (intelligently) unroll loops. The above example, when compiled in GCC with -funroll-loops, will result in eight-way loop unrolling.
It's useful to note, however, that its possible to unroll *too far* which will result in register spilling (and thus slower code).

1.4 Loop Jamming

A very contrived example – consider a function which computes the sum of two arrays. A *terrible* programmer might write something like this:

```
int sum(int a[], int b[], int length){
  int i, sum = 0;
  for(i=0; i<length; i++){
    sum+=a[i];
  }
  for(i=0; i<length; i++){
    sum+=b[i];
  }
}
```

We could clearly condense the above two loops into the same loop:

```
int sum(int a[], int b[], int length){
  int i, sum = 0;
  for(i=0; i<length; i++){
    sum+=a[i] + b[i];
  }
}
```

This (obvious) technique has been given the name *loop jamming*.

1.5 Loop inversion

Programmers typically write loops such that they have some index that starts at `0` and increments until some predetermined size. In assembly, this means we keep incrementing some register and comparing it to some predefined value. Concretely, we'll always need three lines of code:

```
#define SIZE 128

int up(int a[]){
  int i, sum=0;
  for(i=1; i<SIZE; i++){
    sum+=a[i];
  }
  return sum;
}

/*The above generates the following X86 fragment*/

#Assume %ecx holds some our index variable
incl %ecx
cmpl %ecx, SIZE
jne loop
```

The astute programmer will recall that the arithmetic operators set condition codes... there's something else we could do!

```
#define SIZE 128

int down(int a[]){
  int i=SIZE-1, sum=0;
  while(i!=0){
    sum+=a[i];
    i--;
  }
  return sum;
}

/*The above generates the following X86 fragment*/

#Assume %ecx holds some our index variable
decl %ecx
jne loop
```

Though this may not seem like an improvement, observe that we will perform exactly `N` less comparisons (assuming `N` is the length of our array)... that's a huge savings!

1.6 Strength reduction

Consider the fact that many arithmetic expressions have multiple equivalent forms, and that some of those forms can be performed more quickly in assembly than others. As an example, consider the following two equivalent functions:

```
//OLD
void f(){
  x = w % 8;
  y = pow(x, 2.0);
  z = y * 33;
  for (i = 0; i < MAX; i++) {
    h = 14 * i;
    printf("%d", h);
  }
}

//NEW
void g(){
  x = w & 7;           /* bit-and cheaper than remainder */
  y = x * x;           /* mult is cheaper than power-of */
  z = (y << 5) + y;    /* shift & add cheaper than mult */
  for (i = h = 0; i < MAX; i++) {
    printf("%d", h);
    h += 14;         /* addition cheaper than mult */
  }
}
```

That being said, strength reduction is typically performed by the compiler by default, and it almost always results in code that is harder to read... think critically!

1.7 Arithmetic Reordering

This topic is mentioned in book, but we can take it further! The ANSI C standard dictates that arithmetic operations will not be reordered, and this is due to the fact that commutativity does not hold, in general. We (the programmers) can decide to rewrite equations such that we minimize the number of operations that must take place.

```
  //OLD
  float a, b, c, d, f, g;
    a = b / c * d;
    f = b * g / c;

  //NEW
  float a, b, c, d, f, g;
  float div = b / c;
  a = div * d;
  f = div * g;
```

In the above, we've reduced the original form having 2 multiplications and 2 divisions to some form which has 2 multiplications but only 1 division.

2 C Memory Optimizations

Do you want to make your code fit in RAM on a satellite? Do you need *unbelievable* speed, such as writing code to read off a wire? If so, you need to enter the arcane world of byte packing!
To start, recall (or be exposed to) the rules for alignment:

1. chars can start on any byte address
2. 2-byte shorts must start on an even address
3. 4-byte ints or floats must start on an address divisible by 4
4. 8-byte longs or doubles must start on an address divisible by 8

To think about the underlying logic, consider how much extra work the computer would have to do to if it *weren't* the case that things were aligned: the computer might have to access two adjacent words and mask off parts... inefficient!

2.1 Alignment Padding

What does all of this mean, though? Consider the following example:

```
/*Assume 32 bit word size*/
/* Let base address be 0x100 */
char *p; //Self aligned to address divisible by 4... 0x100
char c;  //Put at next free byte... 0x104
int x;   //Put at next address divisible by 4... 0x108
```

Uh oh... what is at memory locations `0x105, 0x106, 0x107`? The technical answer is *slop*. The previous code example could be viewed as containing an implicit padding, like so:

```
char *p;
char c;
char pad[3];
int x;
```

This concept can be taken to the extreme with `struct`'s:

```
struct foo3 {
  char *p;    /* 4 bytes */
  char c;     /* 1 byte */
};

struct foo3 singleton;
struct foo3 quad[4];
```

We can now ask *"What's the stride address of structure* `foo3`*?"* The answer is 8 bytes... even though we only used 5 of them! This means that `quad` contains 12 bytes of slop – but uses 32 bytes total – implying we have 37.5% slop! Using this knowledge, it is possible to reorder your declarations such that you minimize the slop! Keep this in mind!