0. Down the Rabbit Hole

Up until now, we’ve treated the compiler as a magical black box that converts your C code into assembler. You might know about some of the flags you can pass to your compiler, like `-O2`, which turns on many compiler level optimizations, which can make a huge difference in the runtime of your program.

The ultimate goal of a compiler optimization is to calculate exactly what the programmer wants to calculate, while maybe doing so in a completely different way. So long as the optimized assembly is indistinguishable from the program the programmer wrote, the optimizer is free to do whatever it likes to your code.

These days, compilers are good enough to make many “classic” optimizations without your assistance. Techniques such as constant folding, loop unrolling, strength reduction, and inlining can be done automatically, and while the subject is certainly interesting (take CS 153!), in practice it’s more important to know the limitations of what the compiler can do.

Let’s take a look at a few potential optimizations and “optimization blockers” that prevent the compiler from Doing The Right Thing (tm).

1. Loop Invariant Hoisting

Loop invariant hoisting is an optimization where you can take a value that doesn’t change in the loop body and lift it outside of the loop body.

Consider the following code:
void str_to_lower(char *str) {  
    int x = strlen(str);  
    for (int i = 0; i < strlen(str); i++) {  
        if (str[i] >= 'A' && str[i] <= 'Z') {  
            str[i] -= ('A' - 'a');  
        }  
    }  
}  

QUESTION: Why is this code slow?

We call strlen once every loop iteration, and strlen takes O(n) time. This function takes O(n^2) time.

QUESTION: How can we fix it?

Save the result of strlen in a temporary variable:

void str_to_lower(char *str) {  
    for (int i = 0, len = strlen(str); i < len; i++) {  
        if (str[i] >= 'A' && str[i] <= 'Z') {  
            str[i] -= ('A' - 'a');  
        }  
    }  
}  

QUESTION: Why didn’t the compiler do this for us?

If we didn’t modify str in the body of the loop, the compiler would have detected this and “hoisted” the call to strlen out of the body of the loop. In this case, however, the compiler can’t tell that we’re not changing the length of the string by adding ‘\0’ bytes, so it can’t “cache” the length of a string in a temporary variable.

Actually, there is enough information in the program above for the compiler to prove this for us, but neither of gcc and clang do this in practice. Why do you think this is?

2. Pointer Aliasing
Pointer aliasing is when two different pointers might point to the same region of memory. Because the compiler doesn’t know when this can happen, the optimizer can only perform certain “safe” reorderings and optimizations on your code.

Consider the following example:

```c
void prefix_sum(int *array, int *sums, size_t n) {
    if (n == 0) return;
    sums[0] = array[0];
    for (size_t i = 1; i < n; i++) {
        sums[i] = sums[i-1] + array[i];
    }
}
```

**QUESTION:** Why is this code slow? (or, slower than it needs to be)

We have to make two memory loads and one store every time through the loop.

**QUESTION:** How can we fix it?

We can cache the previous partial sum in a temporary variable instead of fetching it from our array:

```c
void prefix_sum(int *array, int *sums, size_t n) {
    if (n == 0) return;
    int sum = sums[0] = array[0];
    for (size_t i = 1; i < n; i++) {
        sum += array[i];
        sums[i] = sum;
    }
}
```

Now we’re only doing one load and one store per loop iteration.

**QUESTION:** Why didn’t the compiler do this for us?

We have one additional piece of information: that `array` and `sums` must point to different bits of memory in order for this function to make sense. The compiler can’t make this assumption in general, however. Consider the following example:

```c
int a[5] = {1, 2, 3, 4, 5};
prefix_sum(a, a+1, 4);
```
In this case, the original program will produce \(a = \{1, 1, 2, 4, 8\}\), which is different than our “optimized” version, \(a = \{1, 1, 3, 6, 10\}\). Since these are different outputs, the “optimized” version is different than the source program, and this is an invalid optimization for the compiler to make.

A simpler example of pointer aliasing is:

```c
void double1(int *number, int *result) {
    *result += *number;
    *result += *number;
}

void double2(int *number, int *result) {
    *result += *number * 2;
}
```

**QUESTION:** Why aren’t the functions above equivalent?

They do different things when `number == result`. If the parameters are the same value, then `double1` quadruples `*result`, and `double2` triples `*result`.

### 3. Prefetching

The compiler is smart enough to handle many of the optimizations you might think to do, and the many layers of CPU caching that you learned about last week give you good memory performance on many common applications.

There are, however, times in which you, the programmer, **know** what the program is going to do next, which the CPU and the compiler simply can’t predict. Consider the following example:

```c
struct int_list {
    int n;
    struct int_list *next;
};

void play_with_list(struct int_list *list) {
    while (list != NULL) {
```
QUESTION: How can we make this code better?

Since we (almost) always fetch the next item in the list, we could advise the CPU that we’re going to access list->next before we call do_something. That way, by the time we finish with that function, we will have already loaded the next entry into the CPU’s cache. For example,

```c
void play_with_list(struct int_list *list) {
    while (list != NULL) {
        __builtin_prefetch(list->next);
        do_something(list->n);
        list = list->next;
    }
}
```

QUESTION: Can you think of other situations in which prefetching might be helpful?

- Tree traversal (prefetch left and right children)
- Matrix multiplication (prefetch the next row while you multiply the current one)
- Other things where you have predictable but random data structure traversal (i.e., pointer following), since it’s trickier for the CPU to predict

Prefetching is something you should add only when you know you need it. Most modern CPUs (like the Intel i7) do some amount of prefetching on its own based on your data access patterns, particularly if you’re accessing memory sequentially with a small stride.