CS61 Section 8 Notes

Week 11/12-11/16

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) {
    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?

QUESTION: How can we fix it?

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

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:

```
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)

QUESTION: How can we fix it?

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

A simpler example of pointer aliasing is:

```
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?

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:

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

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

QUESTION: How can we make this code better?

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

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.