Lecture 19 - Time: Speed and Optimizations (optional) (last updated: 01/27/23)

Last lecture, we explored some aspects of memory allocation and management through the lens of the C programming language. C is a great language for such discussions because its memory model closely follows that upon which compiled programs are built. With C, it is easy to precisely describe the content and location of in-memory data.

In contrast to its spatial acuity, C turns out to greatly conceal the timing properties of a program's behavior: the language says nothing about how fast or slow code should execute. While this ambiguity opens up fruitful opportunities for program optimization, it also leads to a lot of unpredictability.

In this lecture, we will explore the implications of C's timing guarantees (or lack thereof), and discuss some avenues of performance engineering for low-level code. Our discussion will revolve around three sources of timing uncertainty in computer programs.

High-level optimizations

Take a guess: how long (in microseconds) will the following function take to execute and return?

```
// Sum the integers from 1 to 1 million
long foo(void) {
    long v = 0;

    for (long i = 1; i <= 1000000; i++)
        v += i;

    // The value of v is 500000500000
    return v;
}
```

That's a trick question! It's totally under-specified: I haven't given you any information about the speed of the computer you're running it on. But it is sometimes useful to get a feeling for the magnitude of concrete runtime values by experimentally validating (or refuting) your estimates.

Now what about the following function:

```
long bar(void) {
    return 500000500000;
}
```

bar() returns the same value as foo(), but the return value is a hard-coded constant. Concrete durations aside, is bar() faster or slower than foo()?

This question is actually still under-specified, because we don't how foo() and bar() are compiled. A "smart" compiler may recognize that foo() always computes the same constant value, and generate machine code that simply returns that value as a constant (like bar()) instead of recomputing it each time. So foo() and bar() might execute in the same amount of time!

It turns out most compilers are quite "smart" and do more than just translate your source code into machine code. Where possible, optimizing compilers will attempt to generate faster machine code, as long as the functional behavior of

that machine code does not deviate from that of your source program. These optimizations take advantage of C's lack of timing guarantees, meaning slow machine code is considered equivalent to fast machine code that computes the same values and (more or less) performs the same memory accesses.

Some compiler optimizations are guaranteed to make your code faster; others are based on heuristics from common programming patterns. Here are some common types of compiler optimizations:

- Constant folding: for code that computes a constant value, compilers can replace that computation with a constant result. For example, gcc will replace 2 + 4 with just 6, or sizeof(int) * 5 with 20. We saw constant folding in our earlier example, when foo() was optimized to return a constant value like bar().

- Code motion: when a compiler sees that the same computation is performed multiple times (e.g., in a loop), it can try to "hoist" (factor out) that repeated computation and reuse the result. For example:

```
int baz(int i, char *str) {
    int v = 0;

    while (i--) {
        // Compute length of str
        int length = 0;
        for (char *t = str; *t; t++)
            length++;

        v += length;
    }
    return v;
}
```

  baz() computes str's length i times. But the length calculation can be hoisted out like this:

```
int baz(int i, char *str) {
    int v = 0;

    // Compute length of str
    int length = 0;
    for (char *t = str; *t; t++)
        length++;

    while (i--)
        v += length;

    return v;
}
```

  Now baz() only needs to calculate str's length once.

- Function inlining: calling a function takes time! Though the overhead is very small, it can be come relatively significant if your code calls the same short function many times:

```
int add(unsigned int a, unsigned int b) {
    return a + b;
}
```

```
int mul(unsigned int a, unsigned int b) {
    for (int i = 0; i < b; i++)
        a = add(a, a);
    return a;
}
```

We can eliminate the overhead of mul() calling add() by inlining it:

```
int mul(unsigned int a, unsigned int b) {
    for (int i = 0; i < b; i++)
        a = a + a;
    return a;
}
```

These kinds of simple optimizations work best when combined altogether. For example, by inlining a function, you may reveal an opportunity for constant folding. Some optimizations also come at a cost: if you inline functions too aggressively, you risk increasing the size of your executable for negligible performance gain (large executable sizes come with their own performance costs).

Some compilers do not optimize your code by default. For instance, GCC only optimizes your code if you pass the -O compiler flag:

```
$ gcc -c -O1 foo.c -o foo.o
```

The -O flag is actually an alias for -O1 and only enables some optimizations. You can enable more aggressive optimizations using -O2, -O3, and so on, though these usually lead to diminishing returns. It is usually sufficient to stop at -O1 or -O2. In certain cases, over-aggressive optimizations can also make your program take longer to compile and harder to debug.

It can be tempting to optimize your code by yourself, rather than relying on your compiler to do it for you. However, doing so usually comes at the cost of readability and maintainability, and does not always lead to any significant benefit in the grand scheme of things: for instance, if your primary source of inefficiency is due to the latency of reading file contents from disk, then hand-optimizing what you do with that data may not be the best use of your time as a software developer. Instead, you should let a optimizing compiler figure out the best optimization strategy for most code, and only get your hands dirty if you have a performance bottleneck that you know your compiler cannot fix.

Low-level optimizations

There are also optimizations that can't be easily explained using C code because they are related to assembly- and hardware-level considerations. At this low level of abstraction, data can also be stored in CPU registers and caches, in addition to memory. Memory is orders of magnitude slower to access than caches, which is in turn orders of magnitude slower than registers. Compilers, especially with optimizations turned on, will try to take advantage of these access speed differences.

Consider strlen():

```
size_t strlen(const char *s) {
    size_t l = 0;
    while (*s++)
        l++;
    return 0;
```

```
    }
```

Even though the variable l is defined a stack variable in C, the compiler knows that strlen() never asks for the memory address of l. In other words, it doesn't matter whether l is actually implemented using 8 bytes on the stack or an 8-byte register. Recognizing this, a smart compiler will not bother to allocate space for l on the stack and keep it in a register.

The limitation of using registers is that they are in short supply: each CPU only has a fixed number of them, and they are also needed for storing intermediate values. For example, for the following arithmetic expression:

```
    x + y + z
```

most CPUs need to store the result of x + y in a register before adding it to z. Compiles typically first assume a CPU with an infinite number of registers, before later "spilling" register variables to the stack as needed.

Modern CPUs will also try to speed up memory accesses by taking advantage of typical access patterns, and store recently and frequently accessed chunks of memory in high-speed caches. Caches take advantage of the fact that most programs tend to repeatedly access addresses that are close to each other: e.g., if a program reads the first byte of an array, adjacent bytes will also be cached, speeding up access to subsequent bytes of that array.

The speed-ups due to caching are often dramatic enough to overcome the theoretical limitations of using arrays, relative to other data structures. For example, adding an element to the front of an array takes O(n), compared to O(1) for pushing an element to the front of a linked list. However, because array elements are contiguous, the O(n) operation is negligible compared to the overhead of having to heap-allocate a new node for the linked list, as long as n is not pathologically large.

Choosing the optimal data structure in a low-level language like C can be tricky, because it gives you a lot of control over the memory layout of your data without providing much insight on the implications that layout can have on memory access speed. After all, memory access speeds can vary greatly between different hardware architectures and program workloads. Thus, you should always try to collect relevant performance metrics to inform decisions made about the memory layout of your programs.

System overhead

Recall that processes interact with their environment via the operating system, using system calls. Those system calls can incur far more overhead than slow memory accesses or suboptimal control flow. Many libraries go to great lengths to avoid frequent system calls. For instance, getpid() is typically implemented as a library function that looks something like this:

```
    static pid_t pid = 0;               // Initialize to an invalid PID

    pid_t getpid(void) {
        if (pid == 0)
            pid = syscall(SYS_getpid);  // Only perform "actual" system call
                                        // if necessary
        return pid;
    }
```

Here, syscall() is what performs the actual underlying system call, and is what we want to avoid calling unnecessarily. Since a process's PID doesn't ever change, the getpid() function "caches" the actual pid in userspace after the first SYS_getpid system call to avoid making subsequent syscall()s.

The cost of performing system calls is also why FILE output buffering exists. When you call fputs(), fprintf(), or fwrite() on a FILE pointer, the C standard library tries to accumulate as many bytes as it can in a buffer before eventually calling write() to flush that buffer, in hopes of minimizing the number of times write() is called.

When a process performs a blocking system call that cannot be fulfilled immediately, (e.g., read() or accept()), the kernel suspends that process until the system call can be fulfilled (e.g., when a file's contents are retrieved from disk, or when a client connects to a socket). These suspensions incur orders of magnitude more overhead than other sources of latency, so these system calls should be used with great care in performance-sensitive contexts.