Contents

1 — Programming for Performance

Performance!

By this point, I'm certain you know what "programming" means, but we need to take a minute right off the top to define "performance". This course is not about how to program when other people are watching (fun as that can be, as the popularity of Hackathons shows). What it's really about is making a program "fast". Alright, but what does it mean for a program to be fast?

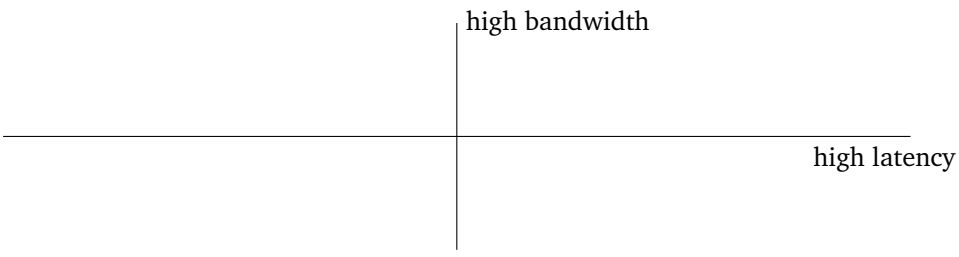
Let's think about the program execution as completion of some number of items—things to do. We have two concepts: items per unit time (bandwidth—more is better), and time per item (latency—less is better). Improving on either of these will make your program "faster" in some sense. In a way they are somewhat related: if we reduce the time per item from 5 s to 4 s it means an increase of 12 items per minute to 15 items per minute... if the conditions are right. Hopefully we could improve both metrics, but sometimes we'll have to pick one.

Items per unit time. This measures how much work can get done simultaneously; we refer to it as bandwidth. Parallelization—or doing many things at once—improves the number of items per unit time. We might measure items per time in terms of transactions per second or jobs per hour. You might still have to wait a long time to get the result of any particular job, even in a high-bandwidth situation; sending a truck full of hard drives across the continent is high-bandwidth but also high-latency.

Time per item. This measures how much time it takes to do any one particular task: we can call this the latency or response time. It doesn't tend to get measured as often as bandwidth, but it's especially important for tasks where people are involved. Google cares a lot about latency, which is why they provide the `8.8.8.8` DNS servers. (Aside relevant to a number of Capstone Design Projects I've seen: when dealing with users, 100ms is the maximum latency for systems that purport to respond instantaneously [Nie93].)

Examples. Say you need to make 100 paper airplanes. What's the fastest way of doing this?

Here's another example, containing various communications technologies:



We will focus on completing the items (doing useful work), not on transmitting information, but the above example illustrates the difference between bandwidth and latency.

Improving Latency

Although we'll mostly focus on parallelism in this course, a good way of writing faster code is by improving single-threaded performance. Unfortunately, there will be a limit to how much you can improve single-threaded performance; however, any improvements here may also help with the parallelized version. On the other hand, faster sequential algorithms may not parallelize as well. But let's take a look at some ways you can improve latency.

Profile the code. You can't successfully make your code faster if you don't know why it's slow. Intuition seems to often be wrong here, so run your program with realistic workloads under a profiling tool and figure out where all the time is going. This is a specific instance of one of my favourite rules of engineering: "Don't guess; measure".

Let's take a quick minute to visit `http://computers-are-fast.github.io/` and take a quiz on how fast computers can do certain operations [EM15]. Are the results surprising to you? Did you do really well or really badly? Chances are that you got some right and some wrong... and the ones that were wrong were not just a little wrong, but off by several orders of magnitude. Moral of the story is: don't just guess at what the slow parts of your code are. It's okay to have a theory as a starting point, but test your theory.

Do less work. A surefire way to be faster is to omit unnecessary work. Two (related) ways of omitting work are to avoid calculating intermediate results that you don't actually need; and computing results to only the accuracy that you need in the final output.

Interesting to note: producing text output to a log file or to a console screen is surprisingly expensive for the computer. Sometimes one of the best ways to avoid unnecessary work is to spend less time logging and reporting. It might make debugging harder, yes, but once the code is correct (or close enough), removing the logging and debugging statements can actually make a difference. Especially in a multithreaded context, logging and debugging often incur synchronization cost.

A hybrid between "do less work" and "be smarter" is caching, where you store the results of expensive, side-effect-free, operations (potentially I/O and computation) and reuse them as long as you know that they are still valid. Caching is really important in certain situations.

Be prepared. If you know something that the user is going to ask for in advance, you can have it at the ready to provide upon request. Example from that other job of mine [JZ]: users often want an Excel export of various statistics on their customs declarations. If the user asks for the report, generating it takes a while, and it means a long wait. If, however, the report data is pre-generated and stored in the database (and updated as necessary) then putting it in the Excel output file is simple and the report is available quickly.

Be smarter. You can also use a better algorithm. This is probably "low hanging fruit" and by the time it's time for P4P techniques this has already been done. But if your sorting algorithm is $\Theta(n^3)$ and you can replace it with one that is $\Theta(n^2)$, it's a tremendous improvement even there are yet better algorithms out there. An improved algorithm includes better asymptotic performance as well as smarter data structures and smaller constant factors. Compiler optimizations (which we'll discuss in this course) help with getting smaller constant factors, as does being aware of the cache and data locality/density issues.

Sometimes you can find this type of improvements in your choice of libraries: you might use a more specialized library which does the task you need more quickly. The build structure can also help, i.e. which parts of the code are in libraries and which are in the main executable. It's a hard decision sometimes: libraries may be better and more reliable than the code you can write yourself. Or it might be better to write your own implementation that is optimized especially for your use case.

Improve the hardware. Once upon a time, it was okay to write code with terrible performance on the theory that next year's CPUs would make it acceptably, and spending a ton of time optimizing your code to run on today's processors was a waste of time. Well, those days seem to be over; CPUs are not getting much faster these days (evolutionary rather than revolutionary change). But sometimes the CPU is not the limiting factor: your code

might be I/O-bound, so you might be able to improve things dramatically by going to solid-state drives or non-volatile memory (e.g. Optane/3DXpoint); or you might be swapping out to disk, which kills performance (add RAM). Profiling is key here, to find out what the slow parts of execution are. When it comes down to it, spending a few thousand dollars on better hardware is often much cheaper than paying programmers to spend their time to optimize the code. (Programmers are super expensive.)

On using assembly. Not that long ago, compilers were not very smart and expert programmers could outsmart the compiler and produce better assembly by hand. This tends to be a bad idea these days. Compilers are going to be better at generating assembly than you are. Furthermore, CPUs may accept the commands in x86 assembly (or whatever your platform is) but internally they don't operate on those commands directly; they rearrange and reinterpret and do their own thing. Still, it's important to understand what the compiler is doing, and why it can't optimize certain things (we'll discuss that), but you don't need to do it yourself. However, giving hints to the compiler about e.g. vector instructions can be helpful.

Anecdote time. A few years ago, I [JZ] was presented with a ticket that read as follows: "the report generation has been running for three hours; I think it's stuck." Turns out the report had not been running for that long, it reached a 30 minute time limit and the server had killed the task (and it just looked like it was running). So now I have a puzzle: how do I speed up this task to get it under the 30 minute time limit?

How does the report work? It selects the transactions for a given period from the database. Then for each transaction, it looks up the latest article data, recomputes the transaction's worth based on the most up to date currency exchange rate, and then stores the updated transaction in the database again.

Step one was to bring up the profiler and look at a few things. The slow steps were primarily database operations: retrieving of exchange rates, retrieving the article data, and then storing all the transactions. Right, with this data, it's time to apply some strategies here.

Caching played a big role: the exchange rate data doesn't change for the report (it is run retroactively, with a date on the end of the last month, so the exchange rates are defined for that day rather than floating). So retrieving the exchange rate 500 times can be cut down to once per currency. Caching was also important for the articles; an article might be used dozens of times, so loading it from the database repeatedly is also a waste of time. Also, I could select all the articles at once rather than each one as encountered.

How about doing less work? For one thing, instead of pulling all the fields of the article from the database, why not just get the five that are actually needed? And in saving the transactions, what if we only update the parts that changed rather than update the full transaction and all its parts?

Ultimately, these techniques combined brought the report time down under 30 minutes and it can now run to completion.

Doing more things at a time

Rather than, or in addition to, doing each thing faster, we can do more things at a time.

Why parallelism?

While it helps to do each thing faster, there are limits to how fast you can do each thing. The (rather flat) trend in recent CPU clock speeds illustrates this point. Often, it is easier to just throw more resources at the problem: use a bunch of CPUs at the same time. We will study how to effectively throw more resources at problems. In general, parallelism improves bandwidth, but not latency. Unfortunately, parallelism does complicate your life, as we'll see.

Different kinds of parallelism. Different problems are amenable to different sorts of parallelization. For instance, in a web server, we can easily parallelize simultaneous requests. On the other hand, it's hard to parallelize a linked list traversal. (Why?)

Pipelining. A key concept is pipelining. All modern CPUs do this, but you can do it in your code too. Think of an assembly line: you can split a task into a set of subtasks and execute these subtasks in parallel.

Hardware. To get parallelism, we need to have multiple instruction streams executing simultaneously. We can do this by increasing the number of CPUs: we can use multicore processors, SMP (symmetric multiprocessor) systems, or a cluster or machines. We get different communication latencies with each of these choices.

We can also use more hardware, like vector processing units built into all modern chips (SIMD) or graphics processing units (GPUs).

Difficulties with using parallelism

You may have noticed that it is easier to do a project when it's just you rather than being you and a team. The same applies to code. Here are some of the issues with parallel code.

First, some domains are "embarassingly parallel" and these problems don't apply to them; for these domains, it's easy to communicate the problem to all of the processors and to get the answer back, and the processors don't need to talk to each other to compute. The canonical example is Monte Carlo integration, where each processor computes the contribution of a subrange of the integral.

I'll divide the remaining discussion into limitations and complications.

Limitations. Parallelization is no panacea, even without the complications that I describe below. Dependencies are the big problem.

First of all, a task can't start processing until it knows what it is supposed to process. Coordination overhead is an issue, and if the problem doesn't have a succinct description, parallelization can be difficult. Also, the task needs to combine its result with the other tasks.

"Inherently sequential" problems are an issue. In a sequential program, it's OK if one loop iteration depends on the result of the previous iteration. However, such formulations prohibit parallelizing the loop. Sometimes we can find a parallelizable formulation of the loop, but sometimes we haven't found one yet.

Finally, code often contains a sequential part and a parallelizable part. If the sequential part takes too long to execute, then executing the parallelizable part on even an infinite number of processors isn't going to speed up the task as a whole. This is known as Amdahl's Law, and we'll talk about this in a few weeks.

Complications. It's already quite difficult to make sure that sequential programs work right. Making sure that a parallel program works right is even more difficult.

The key complication is that there is no longer a total ordering between program events. Instead, you have a partial ordering: some events A are guaranteed to happen before other events B, but many events X and Y can occur in either the order XY or YX. This makes your code harder to understand, and complicates testing, because the ordering that you witness might not be the one causing the problem.

Two specific problems are data races and deadlocks.

- A *data race* occurs when two threads or processes both attempt to simultaneously access the same data, and at least one of the accesses is a write. This can lead to nonsensical intermediate states becoming visible to one of the participants. Avoiding data races requires coordination between the participants to ensure that intermediate states never become visible (typically using locks).
- A *deadlock* occurs when none of the threads or processes can make progress on the task because of a cycle in the resource requests. To avoid a deadlock, the programmer needs to enforce an ordering in the locks. Or use some other strategies as we have discussed in previous courses.

Another complication is stale data. Caches for multicore processors are particularly difficult to implement because they need to account for writes by other cores.

Scalability

It gets worse. Performance is great, but it's not the only thing we're interested in. We also care about *scalability*: the trend of performance with increasing load. A program generally has a designed load (e.g., we are expecting to handle x transactions per hour). A properly designed program will be able to meet this intended load. If the performance deteriorates rapidly with increasing load (that is, the number of operations to do), we say it is *not scalable* [Liu09]. This is undesirable, of course, and for the most part if we have a good program design it can be fixed. If we have a bad program design, then no amount of programming for performance techniques are going to solve that ("rearranging deck chairs on the Titanic").

The things we're going to look at in this course are ways to meet x or even raise the value of x. Even the most scalable systems have their limits, of course, and while higher is better, nothing is infinite. There's only so much we can do to push it, but chances are we can make some serious progress if we make the effort.

Laws of Performant Software

Suppose you want to write fast programs and you like checklists and handy rules. If so, you are in luck, because there is Crista's Five Laws of Performant Software [Lop16].

1. Programming language $\ll$ Programmers' awareness of performance. There is no programming language that is magic, whether good or evil. All the major programming languages allow you to write programs that perform well or badly.

There's a lot of C-elitism in the world, and then there's the back-in-my-day-sonny people who claim assembly was best, and they also had to walk to school in the snow, uphill both ways. High level languages give you lots of options... Do I use an array? A vector? A list? And yes, some of the fancy tools you get are syntactic sugar: they are convenient from the programmer's point of view, but what do they do behind the scenes? If the performance is not what you expect, there is probably be a better way to do it in the high level language.

I'll add my own asterisk on this rule: some languages lend themselves better to parallelization than others. A language may force a certain way of thinking, based on its rules (e.g., functional programming languages). But there is no reason why the way of thinking can't be applied in another language.

2. $d(f^\tau(x), f^\tau(y)) > e^{\alpha\tau}d(x, y)$ or small details matter. This complicated formula is from the butterfly effect (chaos theory). If two versions of the code are x and y, the difference between the performance outcomes $f(x), f(y)$ is much larger than the difference between the code.

A small code change can have a huge impact. Did you fix a memory leak? The addition of one `free()` call is a single line code change but can, in the long run, have a dramatic impact on performance. Is caching used properly? Can you use a faster serialization algorithm?

Basically: don't overlook the small stuff. It's tempting to think that huge major architectural changes are the solution to everything; but there are plenty of gains to be found in the small things.

3. corr(performance degradation, unbounded resource usage) > 0.9. There is a very high correlation between performance degradation and unbounded use of resources. Often times we focus on functionality: the software must have the following 847 251 features! But if you want a program that scales you need to think in terms of operation, not functionality.

Resources need to be limited. If there aren't hard limits, eventually a resource will be exhausted. If the program starts threads, use a thread pool and the thread pool should have a fixed size. Is there a cache? It needs a maximum size. If you need to read input, don't use a function that reads an entire line (of arbitrary length). Furthermore your program needs design effort given to what happens when resources are exceeded. So you decide to set a request queue size; once that queue is full, further requests are rejected in some well-understood manner.

4. Performance improvements = log(controlled experiments) If you want your code to be faster you have to know why it is slow. It's okay not to know the answers, but not knowing how to find out is a problem. Don't guess;

measure.

5. N*bad != good. No amount of nodes, cores, memory, etc, will save you from poorly-written code. Throwing more hardware at the problem is expensive and ineffective in the long term. Bad code is still bad no matter how much hardware it runs on.

Rust

Previous courses you have taken have likely used C and C++ as systems languages. ECE 459 used to as well! It's possible that one of those was your first programming language and perhaps even the one you've used the most. The languages themselves have their strengths and weaknesses, of course, but there's no denying that these languages come without some of the niceties found in other languages like clever static type checking and garbage collection.

The nature of the languages make it hard, or even impossible, to write code that is fast, correct, and secure. The focus of this course hasn't been on security. But in many cases, writing insecure fast code isn't the right thing. Is it even possible to write secure C and C++?

Maybe not. The usual arguments are something along the lines of experience. Experience isn't it either, given this quotation from Robert O'Callahan: "I cannot consistently write safe C/C++ code.[1]" (17 July 2017) (Holds a PhD in CS from Carnegie Mellon University; was Distinguished Engineer at Mozilla for 10 years; now at Google; etc.)

What about use of better tools and best practices? March 2019: disclosure of Chrome use-after-free vulnerability[2]; 0-day attacks observed in the wild. Google implements best practices, and has all the tools and developers that money can buy!

Much of the advice about how to avoid these problems comes down to "try harder", which is... not helpful. If the strategy is just dragging people and saying that they need to pay more attention, or be more careful, or other similar phrase... this is going to constantly be an uphill battle. Expecting people to be perfect and make no mistakes is unrealistic. What we want here is to make mistakes difficult-to-impossible. (Mitigating the effects of mistakes is another good strategy).

A lot of the problems we frequently encounter are the kind that can be found by Valgrind, such as memory errors or race conditions. Other tools like code reviews and Coverity (static analysis defect-finding tool) exist. These are good, but not perfect. Valgrind, for example, only reports errors that it actually sees executed, so until and unless every function and every code path is run, it might not report a problem. Static analysis tools try to track down problems at compile-time, and that seems like a lot better of a solution.

I like to solve not just an individual problem, but an entire class of problems all at once. A somewhat-recent example: if you change the contents of a list in a background thread while it's being rendered, the rendering thread will fail because the list has changed. I can fix the line of code so the list manipulation does not happen during rendering, and that fixes it once, but not forever: in the future, another person (or even Future Me, having forgotten my previous experience) could write code that calls this function from a background thread. There's no good way (in Java, sadly) to make it so invoking this function incorrectly is a compile-time error, so the best I can do is set a trap in it that throws an error if called inappropriately, so that the responsible developer will find what they did wrong during development and testing. Compile-time error checking is preferable to run-time, because the cost of fixing it is lower if it is caught earlier in the process.

A broader perspective: as you know from your co-op work terms, industrial codebases range from hundreds of thousands to millions (and more) of lines of code. You can fix localized problems, like an object that is allocated in a method, doesn't escape, and isn't freed. Localized problems don't require you to look across large parts of the codebase. But the rendering problem above requires non-local knowledge: you add some code that does list manipulation. While you're doing that, you don't see the requirement to not be called during rendering. You can't keep all of the conventions in your head. Tool support is necessary.

This brings us to Rust. It is an alternative to C/C++, incorporating many good ideas from C++(e.g. RAII, references)

[1] https://robert.ocallahan.org/2017/07/confession-of-cc-programmer.html
[2] https://security.googleblog.com/2019/03/disclosing-vulnerabilities-to-protect.html

and sometimes taking them up a notch. It is a new-school secure systems programming language used by Mozilla's Project Quantum. A design goal of this language is to avoid issues with memory allocation and concurrency. It does so by checking things at compile time that most languages don't check at all, and if so, only at runtime.

Tips and caveats. Like any designed artifact, Rust makes trade-offs. Sometimes the trade-offs will work out in your favour, and sometimes they won't. In particular, you will find some things harder to code in Rust than in C/C++. However, they are also more likely to be correct. Is that worth it? Depends on the context: are you writing throwaway code? A prototype? Code that is destined for production in a critical system?

Even though you've been writing code for at least a couple of years, if you are new to Rust, you will have frustrating moments. It may seem like Rust is out to frustrate you, but Rust's developers have put a lot of effort into producing error messages that try to help. And, as always, please use Piazza and otherwise ask colleagues and course staff for help.

There's the saying "If you know one programming language, you know them all". This saying is true to a first approximation, at least for first-year programming. C/C++/C#/Java are reasonably similar and it's possible to write the same sort of code in at least the object-oriented languages. But the saying is not fully true, for a bunch of reasons. Writing *idiomatic* code in a language is different from writing working code, and I'll encourage you to learn Rust as it is and to write Rust code rather than C++ code masquerading as Rust code. In particular, Rust also admits influences from functional languages like Haskell and OCaml[3]

Rust isn't always faster than C++. We'll talk about a specific example, in the context of exceptions (Rust doesn't have them), in Lecture 3.

The Roadmap

First thing we will do is talk about Rust in some more detail. We learned a little bit about why Rust, but we also acknowledge that it's very likely that you have limited to no experience with the language at all. The intention is not to teach you fundamentals of programming, but instead to guide you on the Rust philosophy so you can apply it in the assignments.

You have a program and you want to make it fast. To understand what's going on we will need some baseline understanding of hardware: architecture, caches, branch prediction. Understanding those will tell you what are the pitfalls that can make your program slow.

An easy way to get a performance boost is parallelizing your program: use threads for a big performance boost, mitigate the risks (with locking) but also do it well.

Then when that's mined out you can start thinking about speculation and also about trying to speed up your single thread performance. You can think about going to OpenCL if you have the right kind of task for it, but conversion is hard and the overhead is large.

If you've done all the things that are sure to make improvement it's time to really dig in with the profiling tools to find where and what to focus on next. And when all (reasonable) improvements have been made, then it's time to make your code work using multiple machines (and apply some queueing theory to find out how many servers you need!).

Acknowledgements

Thanks to Sunjay Varma for general comments as well as specific Rust corrections.

[3] Yes, C++, C# and Java all have incorporated functional features now too, but they're not central to how people write in these languages typically.

2 — Rust Basics

Getting Started with Rust

Rather than just tell you to go off and learn all of Rust on your own, we will spend some time on the subject and tell you about important features and why and how they work towards the goal of programming for performance.

With that said, reading or watching material about a programming language is not a super effective way of learning it. There is really no substitute for actually writing code in the language. For this reason, some optional practice exercises/material is linked in the course resources. They might be trivial, but you'll gain a much better understanding of the subject by being hands-on. You (probably) can't learn to swim from watching videos...

What's *not* here? This isn't intended to cover how to declare a function, create a structure, create an enumeration, talk about if/else blocks, loops, any of that. The official docs explain the concepts pretty well and you'll get used to the constructs when you use them. We need to focus on the main objective of the course without getting sidetracked in how to print to the console.

This material is mostly based off the official Rust documentation [KNC20] combined with some personal experiences (both the good and bad kind).

Semicolons; Many of you are coming from the C/C++/Java world where all statements end with semicolons. In Rust that is not so. Semicolons separate expressions. The last expression in a function is its return value. You can use `return` to get C-like behaviour, but you don't have to.

```
fn return_a_number() -> u32 {
  let x = 42;
  x+17
}

fn also_return() -> u32 {
  let x = 42;
  return x+17;
}
```

Change is painful. Variables in Rust are, by default, immutable (maybe it's strange to call them "variables" if they don't change?). That is, when a value has been assigned to this name, you cannot change the value anymore.

```
fn main() {
  let x = 42; // NB: Rust infers type "i32" for x.
  x = 17;     // compile-time error!
}
```

For performance, immutability by default is a good thing because it helps the compiler to reason about whether or not a race condition may exist. Recall from previous courses that a data race occurs when you have multiple concurrent accesses to the same data, where at least one of those accesses is a write. No writes means no races!

If you don't believe me, here's an example in C of where this could go wrong:

```
if ( my_pointer != NULL ) {
  int size = my_pointer->length; // Segmentation fault occurs!
  /* ... */
}
```

What happened? We checked if `my_pointer` was null? And most of the time we would be fine. But if something (another thread, an interrupt/signal, etc) changed global variable `my_pointer` out from under us we would have a segmentation fault at this line. And it would be difficult to guard against, because the usual mechanism of checking if it is `NULL`... does not work. This kind of thing has really happened to me in production Java code. Put all the if-not-null blocks you want, but if the thing you're looking at can change out from under you, this is a risk[4]

Immutable in Rust is forever (ish). The compiler will not let you make changes to something via trickery. You can ignore a `const` declaration in C by taking a pointer to the thing, casting the pointer, and changing through the pointer. Rust grudgingly permits such dark magicks, but you have to brand your code with the `unsafe` keyword and are subject to undefined behaviour. This unsafe behaviour kinda defeats the point of Rust. (How often is unsafe used? See [AMP+20]).

Of course, if you want for a variable's value to be changeable you certainly can, but you have to explicitly declare it as *mutable* by adding `mut` to the definition, like `let mut x = 42;`. Then later you can change it with `x = 0;`. Our general advice (not speaking for Rust here, just for ourselves) is that you want to minimize the number of times you use this. Still, there are some valid scenarios for using mutation. One is that it might be a lot clearer to write your code such that a variable is mutated; another is that for a sufficiently large/complicated object, it's faster to change the one you have than make an altered copy and have the copy replace the original. Write the best code for your situation. Rust just forces you to make mutability explicit and has the compiler check your work.

Then there are constants, which are different from global variables. Constants are both immutable and immortal: they can never change and they are valid for the whole scope they are declared in. This is how you set program-wide constants that are always available and never change, like `const SPEED_OF_LIGHT_M_S: u32 = 299_792_458;`. They don't really exist at runtime and have no address.

On the other hand, Rust also has global variables, defined using `static`. Such variables are immutable, but they may point to things that mutate, e.g. an `Atomic*` or a `Mutex`. The standard warning about a global variable is that it can be accessed from everywhere, so beware.

Shadowing. Something that isn't really "changing" the variable—but looks a lot like it is—is *shadowing*, which is intended to address the problem of "What do I name this?" In another language you might have a variable `transcript` which you then parse and the returned value is stored in another variable `transcript_parsed`. You can skip that with shadowing, which lets you reuse the original name. An alternative example from the docs:

```
let mut guess = String::new();

io::stdin().read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");
```

In this example, the data is read in as a string and then turned into an unsigned integer. Conceptually, there are two variables, one of type `String` and the other of type `u32`. They just happen to have the same name. The first variable (the one that is shadowed, i.e. the `String` in the example) can no longer be used, which is good to know; i.e. Rust promises that you don't have any aliases to it hanging around.

Memory management

In languages like C, memory management is manual: you allocate and deallocate memory using explicit calls. In other languages like Java, it's partly manual—you explicitly allocate memory but deallocation takes place through garbage collection. C++ supports memory management via RAII, and Rust does the same, but Rust does so at compile-time with guarantees, through ownership, which we'll discuss below.

You might be thinking: what's wrong with garbage collection[5] for this purpose? It is well-understood and lots of

[4] OK, let's hedge a bit. Rust prevents data races on shared memory locations, but not all race conditions—for instance, you can still race on the filesystem. In this case, if `my_pointer` was a global pointer, it would also have to be immutable (because not unique), and then why are we here at all; we wouldn't need to do the check. Aha! But it could be an `AtomicPtr`. Then you can modify it atomically but still get races between the first and second reads, which aren't atomic. More on that later.

[5] Garbage collection also cleans up memory when it's sure that no one is using it anymore—it approximates that by cleaning memory that has no pointers to it. Rust's owned objects, on the other hand, can be cleaned up when the single owner has gone out of scope.

languages use it. Actually, the real answer is the magic word: performance. A language that is garbage-collected has to deal with two things: a runtime, and the actual costs of collecting the garbage.

The runtime thing will seem familiar to you if you've programmed in Java or similar; it is the system that, at run-time, must keep track of what memory has been allocated and when to run the garbage collector and such. There's much more to what the Java runtime (JRE) does, but whatever it does comes with some performance penalty (no matter how small) because its functionality does not come for free.

The other part is that the garbage collection process can be expensive. The newest Java garbage collector is G1[6]. This collector has a concurrent phase which runs alongside your application and cleans up simple trash, along with a parallel phase, which runs in a different thread and may stop the world at times, with probabilistic guarantees on pause times. During such a GC, the garbage collector decides what to keep and what to dispose of, and maybe reorganizes memory. Also, the Garbage Collector can do this (1) whenever it wants, and (2) take as long as it feels like taking. Neither of which is great for performance, or for predictable performance.

Think you're macho and can avoid garbage collection by using C/C++? As we discussed last time, your C/C++ code is probably wrong. Well, mine is anyway, I would really prefer to not judge yours. Aside from that, heap allocation and particularly deallocation is still not free even in C/C++. You don't pay garbage collection costs, but you do pay to manage memory-related data structures. Rust makes it easier to allocate some things on the stack rather than the heap, which can in principle improve performance. But, sure, in general C++ and Rust's memory management overhead should be comparable. It's just that you have to go through hoops to write unsafe Rust code.

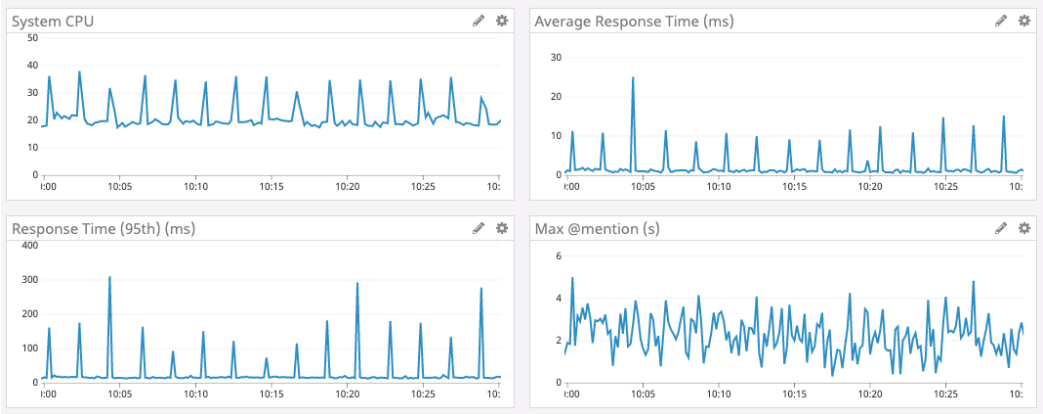
0wn3d

After that memory management discussion, the most important thing to tell you now about Rust is the concept of *ownership*. This strongly distinguishes Rust from other programming languages. Ownership has a number of applications, and lies behind Rust's strategy for memory management and safe concurrency.

Rust uses ownership as a default memory management strategy. That is, the compiler determines (at compile-time, of course) when allocated memory can be cleaned up[7]. In brief, memory can be cleaned up when no one needs it anymore. Ownership imposes certain restrictions on how you write your code and will inevitably cause at least one moment where you angrily curse at the compiler for its refusal to let you do what you want. The compiler is only trying to help. Promise.

The advantage of ownership over RAII is precisely due to the compiler's meddling. You can't mess up and leave a dangling reference around.

Real-World Example: Discord. If you want a real-world example of this, consider this graph from an article about a service at Discord [How20]:

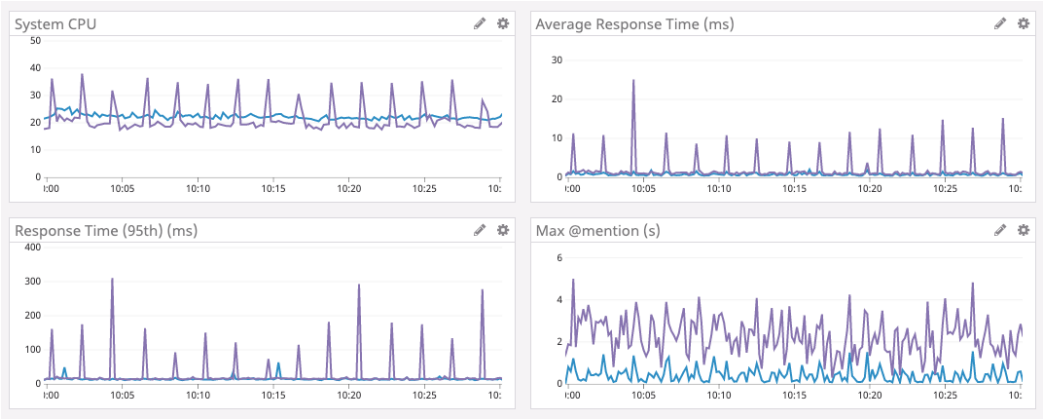


[6] `https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html`

[7] This is a little white lie, but a harmless one; deallocation might be compile-time conditional, with the compiler inserting code to deallocate. See `https://doc.rust-lang.org/stable/nomicon/drop-flags.html`. Also, some things are allocated on the stack.

Quick recap: the Go garbage collector does its work and it adds a big latency spike. Rust would not have those spikes, because of ownership: when memory is no longer needed, it is trashed immediately and there's no waiting for the garbage collector to come by and decide if it can be cleaned up. To be fair, C++ also wouldn't have such spikes (because RAII). The article also adds that even with basic optimization, the Rust version performed better than the Go version. Not only in terms of there being no spikes, but in many dimensions: latency, CPU, and memory usage.

See the following graphs that compare Rust (blue) to Go (purple):



I do recommend reading the article because it goes into some more details and may answer some questions that you have.

The Rules. That long introduction to the concept of ownership didn't explain very much about how it actually works; it just went into the *why* and how it relates to the objectives of this course. But the rules are pretty simple—deceptively so—and they are as follows:

1. Every value has a variable that is its owner.
2. There can be only one owner at a time.
3. When the owner goes out of scope, the value is dropped.

These rules draw a distinction between the value itself and the variable that owns it. So in a statement of `let x = 42;` there is memory associated with the value "42". That memory is the "value" in rule 1, and its owner is the variable `x`.

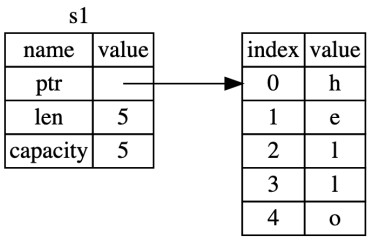
When `x` goes out of scope, then the memory will be deallocated ("dropped"). (This is very much like the RAII (Resource Acquisition Is Initialization) pattern in languages like C++). Variable scope rules look like scope rules in other C-like languages. We won't belabour the point by talking too much about scope rules. But keep in mind that they are rigidly enforced by the compiler. See a brief example:

```
fn foo() {
  println!("start");
  { // s does not exist
    let s = "Hello␣World!";
    println!("{}", s);
  } // s goes out of scope and is dropped
}
```

The same principle applies in terms of heap allocated memory (yes, in Rust you cannot just pretend there's no difference between stack and heap, but ownership helps reduce the amount of mental energy you need to devote to this). Let's learn how to work with those! The example we will use is `String` which is the heap allocated type and not a string literal. We create it using the

```
fn main() {
  let s1 = String::from("hello");
  println!("s1␣=␣{}", s1);
}
```

A string has a stack part (left) and a heap part (right) that look like [KNC20]:



This makes it a bit clearer about what is meant when the rules say that when the owner (the stack part) goes out of scope, the value (the heap part) is deallocated.

That covers rules one and three…But that second rule is interesting, because of the "at a time" at the end: it means that there exists the concept of transfer of ownership.

In fact, everything we've said so far could also be true for much C++ code using RAII: there is an owner for each value, there is only one owner, and things are freed when they go out of scope. However, "there is only one owner" isn't actually enforced by the C++ compiler, so you can write code that breaks it, and then that code is prone to segfaults.

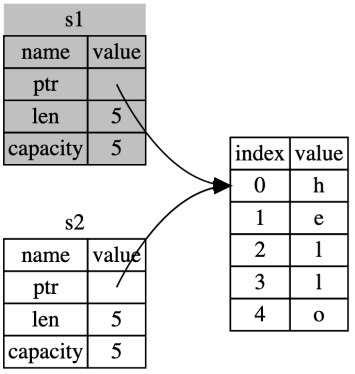
What's yours is mine. Move semantics have to do with transferring ownership from one variable to another. But ownership is overkill for simple types[8] (see the docs for a list—stuff like integers and booleans and floating point types), and such types don't need to follow move semantics; they follow copy semantics. Copy semantics are great when copies are cheap and moving would be cumbersome. So the following code creates two integers and they both have the same value (5).

```
fn main() {
    let x = 5;
    let y = x;
}
```

But simple types are the exception and not the rule. Let's look at what happens with types with a heap component:

```
fn main() {
  let s1 = String::from("hello");
  let s2 = s1;
}
```

Here, no copy is created. For performance reasons, Rust won't automatically create a copy if you don't ask explicitly. (You ask explicitly by calling `clone()`). Cloning an object can be very expensive since it involves an arbitrary amount of memory allocation and data copying. This point is a thing that students frequently get wrong in ECE 252: when doing a pointer assignment like `thing* p = (thing*) ptr;`, no new heap memory was allocated, and we have `p` and `ptr` pointing to the same thing. But that's not what happens in Rust [KNC20]:



[8] Specifically, types with the `Copy` trait have copy semantics by default; this trait is mutually exclusive with the `Drop` trait. `Copy` types have known size at compile time and can be stack-allocated.

If both `s1` and `s2` were pointing to the same heap memory, it would violate the second rule of ownership: there can be only one! So when the assignment statement happens of `let s2 = s1;` that transfers ownership of the heap memory to `s2` and then `s1` is no longer valid. There's no error yet, but an attempt to use `s1` will result in a compile-time error. Let's see what happens.

```
fn main() {
  let x = 5;
  let y = x;
  dbg!(x, y); // Works as you would expect!

  let x = Vec<u32>::new(); // similar to the std::vector type in C++
  let y = x;
  dbg!(x, y); // x has been moved, this is a compiler error!
}
```

The compiler is even kind enough to tell you what went wrong and why (and is super helpful in this regard compared to many other compilers) [KNC20]:

```
plam@amqui ~/c/p/l/l/L02> cargo run
   Compiling move v0.1.0 (/home/plam/courses/p4p/lectures/live-coding/L02)
error[E0382]: use of moved value: ‘x‘
 --> src/main.rs:8:10
  |
6 |     let x = Vec::<u32>::new(); // similar to the std::vector type in C++
  |         - move occurs because ‘x‘ has type ‘std::vec::Vec<u32>‘, which does
  |           not implement the ‘Copy‘ trait
7 |     let y = x;
  |             - value moved here
8 |     dbg!(x, y); // x has been moved, this is a compiler error!
  |          ^ value used here after move

error: aborting due to previous error

For more information about this error, try ‘rustc --explain E0382‘.
error: could not compile ‘move‘.

To learn more, run the command again with --verbose.
```

Move semantics also make sense when returning a value from a function. In the example below, the heap memory that's allocated in the `make_string` function still exists after the reference `s` has gone out of scope because ownership is transferred by the `return` statement to the variable `s1` in `main`.

```
fn make_string() -> String {
  let s = String::from("hello");
  return s;
}

fn main() {
  let s1 = make_string();
  println!("{}", s1);
}
```

This works in the other direction, too: passing a variable as an argument to a function results in either a move or a copy (depending on the type). You can have them back when you're done only if the function in question explicitly returns it!

```
fn main() {
  let s1 = String::from("world");
    use_string( s1 ); // Transfers ownership to the function being called
    // Can't use s1 anymore!
}

fn use_string(s: String) {
  println!("{}", s);
  // String is no longer in scope - dropped
}
```

This example is easy to fix because we can just add a return type to the function and then return the value so it goes back to the calling function. Great, but what if the function takes multiple arguments that we want back? We can `clone()` them all... which kind of sucks. We can put them together in a package (structure/class/tuple) and return that. Or, we can let the function borrow it rather than take it... But that's for next time!

C++ does have move semantics, but it uses copy semantics by default.

Do the Rules Work? With a stronger understanding of the rules and their practicalities, the obvious question is: do they work? There's no point in having the rules if they don't accomplish the goal. We'll assume for the moment that there are no bugs in the compiler that violate the expected behaviour. And then let's consider this from the perspective of some things that can go wrong in a C program.

(Okay, alright, before we get there—we'll eventually learn to break rules and to use reference counted objects where if we get it wrong we can leak. We briefly discuss `Rc` and `Arc` in Lecture 4.)

- Memory leak (fail to deallocate memory)—does not happen in Rust because the memory will always be deallocated when its owner goes out of scope.
- Double-free—does not happen in Rust because deallocation happens when the owner goes out of scope and there can only be one owner.
- Use-after-free—does not happen in Rust because a reference that is no longer valid results in a compile time error.
- Accessing uninitialized memory—caught by the compiler.
- Stack values going out of scope when a function ends—the compiler will require this be moved or copied before it goes out of scope if it is still needed.

A free lunch?

To provide a somewhat balanced view, Rust of course doesn't solve every problem in the world. It does solve memory management well, and I wouldn't even say that it requires more of the programmer: it requires more of the programmer at compile-time, not at debug-time.

Let's revisit garbage collection. If you want to implement graphs and doubly-linked lists[9], GC is really handy. (Many Rust people will argue that you shouldn't use linked lists if you want performance anyway. If you think back to your architecture course, you can deduce why.) Or you can use pointers and unsafe Rust and hope to get it right.

Here are some downsides to consider:

- Static typing: To expand on the point above: there is a New Zealand saying "she'll be right". It's a bit hard to explain, but Wikipedia suggests: "a situation or object which is not perfect but is good enough to fulfil its purpose". Static typing, and Rust, discourage this point of view. The code really does have to satisfy type safety and memory safety properties before it will run.
- Ecosystem: Rust does come with a package registry (crates), which is better than C++, but some libraries are not going to exist in Rust. We'll talk more about calling foreign functions in Lecture 5.
- Compiler: Rust's compiler can be slow on large codebases.

[9] `https://rust-unofficial.github.io/too-many-lists/`

3 — Rust: Borrowing, Slices, Threads, Traits

Borrowing and References

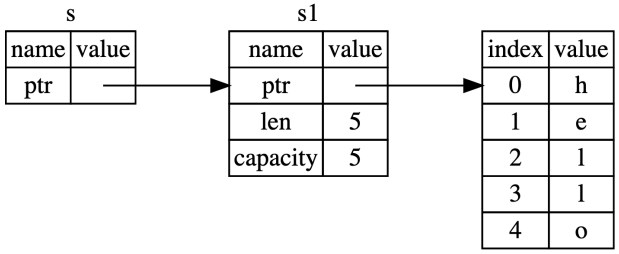
We've already seen that ownership is a concept in Rust that can come with a couple of unintended consequences, e.g. from accidentally giving an argument to a function that we still need later. Rust supports "borrowing"—you need to use the data for something but you also promise you'll give it back (and the compiler forces you to live up to your promises). Borrowing allows data to be shared, but the sharing has to be done in a controlled and safe way to prevent leaks and race conditions.

Rust's compiler analyzes all the borrowing that takes place in the program using the *borrow checker*. If the borrow checker is not certain that your code is perfectly safe, it will say no (and produce a compile time error). This can be a little bit frustrating, because the analysis is not perfect and errs on the side of caution. Eventually we will introduce some ways that you can tell the borrow checker that you guarantee the code is safe, but you have to be sure, otherwise all the usual bad things can happen!

The feature that we need for the concept of borrowing is the *reference*. To indicate that you want to use a reference, use the & operator. The reference operator appears both on the function definition and the invocation, to make sure there's no possibility of confusion as to whether a reference is being expected/provided or ownership is to be transferred. Consider this example from the official docs [KNC20]:

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

When we invoke the `calculate_length` function, ownership of the string is not transferred, but instead a reference to it is provided. The reference goes out of scope at the end of the function where it was used, removing it from consideration. A reference is not the same as ownership and the reference cannot exist without the original owner continuing to exist. That is represented in the official docs by this diagram:



And if you borrow something, it's not yours to do with as you wish—you cannot assign ownership of it (move it),

which makes sense because you can't give someone ownership of something you do not own.

By default, references are immutable: if you borrow something, you cannot change it, even if the underlying data is mutable. Attempting to do so will result in—you guessed it—a compile time error, where the compiler tells you that you are trying to change something that's immutable.

Of course, in real life, you would be much more agreeable to letting people borrow your things if there were strong guarantees that it would (1) always be returned and (2) would be returned in the same condition. That would be nice! But until such time as that magical technology is invented, no, you can't borrow my car. Sorry.

Mutable references do exist, but they have to be declared explicitly as such by tagging them as `&mut`:

```
fn main() {
    let mut s1 = String::from("hello");
    let len = calculate_length(&mut s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &mut String) -> usize {
    s.len()
}
```

Mutable references come with some big restrictions: (1) while a mutable reference exists, the owner can't change the data, and (2) there can be only one mutable reference at a time, and while there is, there can be no immutable references. This is, once again, to prevent the possibility of a race condition. These two restrictions ensure that there aren't concurrent accesses to the data when writes are possible. There's also a potential performance increase where values can be cached (including in CPU registers; we'll come to that later) without worry that they will get out of date.

As long as there are no mutable references, there can be arbitrarily many immutable references at the same time, because reads don't interfere with reads and a race condition does not occur if there are only reads.

References cannot outlive their underlying objects. Below is an example from the official docs that will be rejected by the borrow checker, because the reference returned by `dangle` refers to memory whose owner `s` goes out of scope at the end of the function:

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s // returning a thing that no longer exists upon return
}
```

In C this would be a "dangling pointer" (a pointer that's pointing to a location that is no longer valid). I see this kind of error a lot in C programs where someone has stack allocated a structure and then wants to pass it to another thread and does so with the address-of operator. It compiles and might even work sometimes at runtime, but is still wrong (technically, undefined behaviour) and can eventually be exposed as a bug that bites you.

If we actually try to compile the previous code example, the compiler says something about giving the value a lifetime. We'll come back to the idea of lifetimes soon.

Non-Lexical Lifetimes. A more recent improvement to Rust's borrow checking is called non-lexical lifetimes. Consider the small block of code below:

```
fn main() {
    let mut x = 5;

    let y = &x;
    println!("{}", y);

    let z = &mut x;
}
```

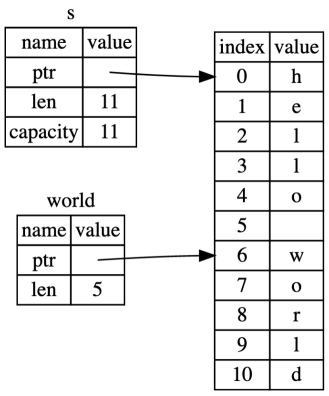
Under the old rules, the compiler would not allow creation of the mutable reference `z` because `y` has not gone out of scope. It would consider `y` to be valid until the end of the function. The improvement of NLL is that the compiler can see that `y` is no longer used after the `println!` macro and hence the `z` reference is okay to create. `y` can be dropped as soon as it's no longer needed; the `z` reference will not exist at the same time; and all is fine.

Slices

The *slice* concept exists in a few other programming languages, and if you have experience with them this will certainly help. A slice is a reference (yes, a reference in the sense of the previous section) to a contiguous subset of the elements of a collection. This is what you do if you need a part of an array (the typical example for that being a substring of an existing string). If our code looks like this:

```
fn main() {
    let s = String::from("hello_world");
    let hello = &s[0..5];
    let world = &s[6..11];
}
```

The representation of the slice looks like [KNC20]:



Slices can also apply to vectors and other collections, not just strings. As with the other kinds of references we've learned about, the existence of a slice prevents modification of the underlying data. Just as with references, slices prevent race conditions on collections but also avoid (as much as possible) the need to copy data (which is slow).

Unwrap the Panic

A quick digression: a lot of functions we use return `Result` types. These return either `Ok` with the type we expected, or `Err` with an error description. To get the type you want, you need to unpack the result.

If we try to open a file but the file doesn't exist, that's an error but one that's foreseeable and we can handle it. There's three ways to handle it: a `match` expression (this is like the `switch` statement), `unwrap()`, and `expect()`.

You may be tempted to just always use `unwrap()` because it gives you the result and calls the `panic!` macro if there's an error. This, however, just shows you the lower level error that is the problem and you are denying yourself the opportunity to add information that will help you debug. For that reason, it's better to use `expect()`, which lets you add your own error message that will make it easier to find out where exactly things went wrong.

It's recommended to use `Result` types for functions you write too. Make your future self happy by giving yourself the information you need to debug what's gone wrong!

This does come at a small performance hit. CS 343 included a performance comparison of exceptions versus error codes (à la `Result`). Exceptions that don't happen (the happy path) are, indeed, faster than explicitly handling error codes / `Result` types. As an engineering trade-off, both exceptions and `Result` types force the programmer to explicitly deal with errors (at the very least, explicitly ignoring them rather than silently ignoring them). Rust

doesn't have exceptions, so you always have to pay the performance hit. (Simulating exceptions with `panic!` is not a best practice.)

Fearless Concurrency

More than just trying to prevent memory problems by making them compiler errors, Rust is also intended to make concurrency errors compile-time problems too! That's actually difficult, of course, but the good news is that the key ideas of ownership and borrowing and such will help you avoid concurrency problems.

The drawback to concurrency is that it brings new problems with it: race conditions, deadlock, that sort of thing. Making your program faster is great, but not if it's at the cost of the answers being incorrect (or your program failing to produce an answer some of the time).

If the compiler can help with making sure your concurrent program is correct, it doesn't make your program faster directly, but it helps indirectly. If you can be (more) sure of the correctness of your code, you don't have to spend as much time testing it before you can deploy it and move on to the next thing. Also, if the bug is prevented from being introduced in the first place, you don't have to spend time debugging it and fixing it, which lets you spend more time on speeding up other things. And honestly, if you are looking at a piece of code that is super business critical, anything that adds to your confidence that no issue has been introduced makes it that much easier to make that change you want to make.

Threads. Rust uses threads for concurrency, with a model that resembles the create/join semantics of the POSIX pthread. If you are unfamiliar with pthreads, the course repository has a PDF refresher of the topic (`pthreads.pdf`). We will talk about the Rust way, but the background material will provide context.

OK, so you want to create a thread! The mechanism for doing so is referred to as spawning a thread. Here's a quick example from the official docs [KNC20]:

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi␣number␣{}␣from␣the␣spawned␣thread!", i);
            thread::sleep(Duration::from␣millis(1));
        }
    });

    for i in 1..5 {
        println!("hi␣number␣{}␣from␣the␣main␣thread!", i);
        thread::sleep(Duration::from␣millis(1));
    }

    handle.join().unwrap();
}
```

A few things make this significantly different from the pthread model that we are used to. First of all, the thread being created takes as its argument a *closure*—an anonymous function that can capture some bits of its environment. The spawn call creates a `JoinHandle` type and that's what we use to call `join`, which is to say, wait for that thread to be finished. As we expect from pthreads, if calling join on a thread that is not finished, the caller waits until the thread is finished.

This is a simple example that works, but fails to capture the complexity of actually working with threads, because there's no data moved between threads. Most interesting uses of threads need some data communication. There are three ways that we can get data from one thread to another: capturing, message passing, and shared state.

Capturing. The notion of "capturing" calls back to the earlier mention that a closure captures some of its environment. That is, the body of the function can reference variables that were declared outside of that function and in the context where `thread::spawn` was called. The compiler will analyze the request and try to figure out what needs to happen to make it work, such as borrowing the value, as in this example (also from the docs):

```
use std::thread;
```

```
fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v); // can't do this
    });

    handle.join().unwrap();
}
```

The only problem is: this example does not work. The compiler is not sure how long the thread is going to live and therefore there's a risk that a reference to `v` held by the thread outlives the actual vector `v` in the main function. How do we fix that?

Well, I had the idea that if I put something after the `join()` call that uses `v`, then the compiler should know that `v` has to remain in existence until after the thread in question. Yet, it still reports the error E0373 that says the thread might outlive the borrowed value. This actually got me thinking about why this didn't work and I decided to ask some of the compiler devs. It has to do with the fact that a thread isn't really a first-class construct in Rust, and the "lifetime" of arguments that you pass has to be sufficiently long. We'll learn about lifetimes soon.

Anyway, the error message suggests what you actually want in this scenario: to move the variables into the thread. To do so, specify `move` before the closure: `let handle = thread::spawn(move || {...` This addition results in the transfer of ownership to the thread being created. You can also copy (i.e. clone-and-move) if you need.

One thing you don't want to do is try to make the lifetime of your vector or other construct `static`, even though the compiler might suggest this. We can revisit that when we talk about lifetimes as well.

Message Passing. Sometimes threads want to communicate in a way that isn't one-way communication at the time that the thread is being created. For that, a possibility is message-passing. This mechanism of communication may seem familiar from previous experience with various UNIX mechanisms like pipes and message queues. This strategy is very structured and generally safer than shared memory, i.e. it is harder to race or to access inappropriate locations.

The ownership mechanic of message passing is like that of postal mail. When you write a physical letter and mail it to someone, you relinquish your ownership of the letter when it goes in the mailbox, and when it is delivered to the recipient, the recipient takes ownership of that letter and can then do with it as they wish.

So you want to have two threads communicate. The metaphor that Rust (and many others) use for this is called a *channel*. It has a transmit end (where messages are submitted) and a receive end (where messages arrive). The standard model is multiple-producer, single-consumer: that is, lots of threads can send data via the sending end, but in the end it all gets delivered to one place. Think of that like postal mail as well: I can drop a letter to you in any postbox or post office, but they will all be delivered to your mailbox where you collect them in the end.

Okay, enough talk, let's make one [KNC20]:

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

The channel constructor returns a tuple with the transmitting end `tx` and receiving end `rx`. We'll then send the transmitting end into the thread and have it send a message to the main thread. The main thread will wait until the message is there and then get it. This does mean that `recv()` is blocking and there is a corresponding `try_recv()`

which is nonblocking. You may have already covered nonblocking I/O in a previous course; if not, we will return to that subject soon.

If you want to have multiple transmitting ends, you need only use `clone` on the transmitter and hand those out as needed.

As a small technical note, the type you want to send via a channel has to implement the `Send` trait (think of traits being like interfaces). Almost all basic types in Rust have this trait, and any programmer-defined type that is composed entirely of types that have it will also have that trait.

Traits

Okay, we have to take a detour here onto the subject of Traits. As the previous paragraph said, traits are a lot like interfaces. You specify a trait as a set of function signatures that you expect that the type in question to implement. A very simple trait and its usage are shown below:

```
pub trait FinalGrade {
    fn final_grade(&self) -> f32;
}

impl FinalGrade for Enrolled_Student {
    fn final_grade(&self) -> f32 {
        // Calculation of average according to syllabus rules goes here
    }
}
```

A couple of other notes about traits are worth mentioning. One, you can only define traits on your own types, not on external (from other packages/crates) types, so that you don't break someone else's code. Two, you can add a default implementation to the trait if you want (something Java lacked for a long time). Third, as in other languages with interfaces, a trait can be used as a return type or method parameter, so it is a kind of generic. Finally, you can use + to combine multiple traits (which is nice when you need a parameter to be two things)

With the preamble out of the way, there are three traits that are really important to us right now. They are `Iterator`, `Send`, and `Sync`.

Iterator is the easiest one to explain. You put it on a collection and it allows you to iterate over the collection. Moreover, this is often more efficient than a typical for loop construction, because it lets the compiler skip over bounds checking and other such issues. Nice.

Send was already introduced. It's necessary to transfer ownership between threads. There are some Rust built-in or standard-library types that very specifically choose not to implement this interface to give you a hint that they are not intended for this purpose. If the compiler tells you no, it's a hint that you want to use a different type. As previously mentioned, if your programmer-defined type is made entirely of types that have the Send trait, then it too has the trait. If you really must use something that is inherently not safe to send, though, you can implement this trait on your type manually and guarantee the thread-safe transfer of ownership yourself, but it's not a good idea if you can avoid it.

Sync is the last one, and it means that a particular type is thread-safe. That means it can be referenced from multiple threads without issue. The primitive types have this trait, as do any programmer-defined types that are composed entirely of Sync types[10]. It's important to just mention here that this does not mean all operations on a Sync type are safe and that no race conditions are possible; it just means that *references* to the type can be in different threads concurrently, and we can't have multiple mutable references. No, if we want more than one thread to be able to modify the value, we need mutual exclusion...

Back to the Mutex...

If you don't want to use message passing for some reason (and performance is a reason, if it's borne out by your testing/data) then there is fortunately the ability to use a mutex for mutual exclusion. We know how these work,

[10] If you are yourself implementing something that implements Sync—not by composition—and you do it wrong, you can cause undefined behaviour. But if you are doing that you will be forced to tag your implementation with the `unsafe` keyword, which is beyond the scope of this course.

so let's skip the part where I make some analogy about them.

What's different about the mutex in Rust is that the Mutex wraps a particular type. So it is defined as `Mutex<T>` and if you want an integer counter initialized to 0, you create it as `Mutex::new(0);`. This way, the mutex goes with the value it is protecting, making it much more obvious what mutex goes with what data, and making it so you have to have the mutex to access the data. A sample from the docs [KNC20]:

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m␣=␣{:?}", m);
}
```

In addition to forcing you to acquire the mutex before you can make any use of the internal value, the lock is automatically released when the `num` variable goes out of scope; the type of `num` is a `MutexGuard` which is our "possession" of the lock; when that possession ends, the mutex is automatically unlocked. This means you want, generally, to use the manual-scoping `{` and `}` braces to ensure that the lock is released when you're done with it and not just at the end of the function or loop.

The use of the mutex in the above program is obviously unnecessary, since there's only the one thread. If we want to use it in multiple threads, we need multiple threads to access it. But we can't, unfortunately, just say that references will do! The mutex type has to outlive the other threads and such and the compiler will suggest moving it... But we can't move it into more than one thread, because that violates our rule about having only one owner. What now?

It looks like we have to break a rule: we need the ability to share ownership of some memory. We don't know how to do that, but when we start with breaking rules, we might find that we like it and might break more than one...

4 — Rust: Breaking the Rules for Fun and Performance

Mutual Exclusion and Multiple Ownership

Mutex and Reference Counting Where we left off previously, we've identified that a mutex is not super amenable to our model of single ownership because we need multiple threads to have access to this mutex. There is a way to do it, but we have to break the single ownership rule, and that requires a little more background on smart pointers and reference counting.

We know what pointers are from C and C++, and if you have sufficient experience with C++ you will know that smart pointers exist in that language too! We'll talk about two kinds of smart pointer right now, the Box and the Reference-Counting type.

The `Box<T>` is an easy way to put some data on the heap rather than the stack. This is good for a situation where you, for example, take input from a user and you don't know in advance how big it's going to be, or when you have some data that you want to transfer ownership of rather than copy (for performance reasons, obviously). You create a Box with `Box::new(...)` as expected, and it's heap allocated with all the usual things that come with it in Rust, like ownership and that it gets dropped if the owner goes out of scope.

The reference counted smart pointer, however, is the thing that allows for shared ownership. There are some reasons why we might want this, even in a single-threaded program, such as a graph data structure. But the main idea is that you can share ownership as much as you like, and the value only goes away when the last reference to it is dropped (reference count goes to zero).

To make a reference-counted object, use type `Rc<T>`. Instantiate that type to get an object; if you want to make another reference to the same object, use `clone()`, which increases the reference count. When references are dropped, the count decreases.

It is important to note that reference types can leak memory! If you've chosen this route for managing data in your program, there is a possibility of forming a cycle in the reference types. If such a cycle is formed, the memory will never be dropped. This is undesirable, of course.

What you can't do, unlike C++'s analogous `shared_ptr`, is keep a reference to the value after the `Rc` or `Arc` goes out of scope. That's because the value is still owned by the pointer and is definitely freed when the pointer goes away.

Right, so we have everything we need now to pass the mutex around, right? Well, almost. `Rc<T>` won't work when we try to pass it between threads, because the compiler says it cannot be sent between threads safely, because it does not implement the special `Send` trait. This is because the management of its internal counter is not done in a thread-safe way. If we want that, we need the *atomic* reference counted type, which is `Arc<T>`. It is perhaps slightly slower than the regular reference counted type, so you won't want to choose it in every scenario, but it's exactly what we need here.

Here's an example of using an atomic reference counted type for setting up a handler for the Ctrl-C (SIGINT); this is modified from a program I wrote that listens for connections and spawns threads if a client connects:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};

fn main() {
   let quit = Arc::new(Mutex::new(false));
    let handler_quit = Arc::clone(&quit);
    ctrlc::set_handler(move || {
        let mut b = handler_quit.lock().unwrap();
        *b = true;
    }).expect("Error setting Ctrl-C handler");

    while !(*quit.lock().unwrap()) {
            // Do things
    }
}
```

In this example, I use a mutex to protect a boolean that's used concurrently (even if it's not in two threads): once in main and once in the handler.

We should also still remember that there exists the possibility of a deadlock in Rust, even if the mutex is automatically unlocked for us. Nothing prevents thread 1 from acquiring mutex A then B and thread 2 from concurrently acquiring B then A. This language cannot solve all concurrency problems, unfortunately.

Lifetimes

We've covered the idea that in Rust, the compiler can make a determination about how long a particular piece of data will live. How long it lives is sometimes referred to as the reference's lifetime. The good news is that the compiler is usually able to make a determination about how long things should live. This system is not perfect, and sometimes we have to help it a bit.

Here's a simple program in the official docs that won't compile because the type system can't figure out what's correct [KNC20]:

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

The compiler says it can't figure out whether the return value is the borrowing of `x` or `y` and therefore it's not sure how long those strings live. It might look like it's obvious at this point, because the two strings are known at compile time. The compiler, however, makes decisions based on local information only (that is, what it finds in the current function it is evaluating). For that reason, it treats `longest` as if it could take any two string references. Alright, that's fine for now, because it was an example to show what happens when we can't know the answer at compile time anyway.

To get this to compile, we have to specify lifetimes using annotations. Annotations don't change how long references live, really. They just describe the relationships between the lifetimes of references. This is used on functions to specify what they can accept and what they can return.

If you'd like an analogy, think of it as saying something like "I will only buy eggs that have an expiration date that is at least two weeks in the future.". This rule does not change the eggs that are in the store. It does not mean that eggs that have an expiration date of next week are poison and nobody should eat them. I'll happily buy eggs that expire in a month. So we are just being clear about what we want here.

Lifetime annotations are written with an apostrophe `'` followed by a name, and names are usually short like `'a` or `'b`. Let's correct the `longest` function:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This does what we need! The first appearance of our lifetime annotation `'a`, after the name of the function, says that all parameters and return value must have the same lifetime. Then we say we will accept strings that live at least as long as our designated `'a` lifetime. That is, it's got to live at least as long as the smallest of `x` and `y`. The actual lifetime isn't as important, all that matters is that it follows the rule of being at least that long.

In early versions of Rust, lifetime annotations had to be specified everywhere. That was somewhat annoying. Fortunately the compiler can identify a lot of common scenarios, so the borrow checker can now read them in where they're needed most of the time.

But we're not breaking rules here, we're applying more rules. What gives? The rule-breaking thing is the ability to grant a particular piece of memory immortality. If you specify as a lifetime the special one `'static`, you grant the ability for this memory to live the entire duration of the program. Just because it can doesn't mean it necessarily will live forever—only that it could.

This can be used correctly to tell the compiler that a particular reference will always be valid, such as string literals that are always going to hang around. It's also used in the interface for spawning a thread, incidentally, which happens because you can pass *anything* to a thread, and the compiler wants to be sure that whatever you are providing is definitely going to live long enough for the thread which can live an arbitrarily-long life (threads are estimated to be immortal).

For the record, the kind of immortality we are talking about here is the Tolkien-Elf kind, where they won't die of old age, but can die in violence or grief. Threads can exit and be cancelled and such in Rust, and static variables can get dropped if the compiler is sure it's safe to do. But they *could* hang around indefinitely.

You can use the static lifetime to bandaid a couple of compiler errors, and the compiler might even suggest it. You shouldn't, though. You should really apply the correct lifetime annotations or fix the would-be dangling reference. The compiler can lead you down the wrong path if it says that it wants a vector being passed to a thread to be annotated as static. What you might think is that it means you should annotate the function parameter with this lifetime. But that just moves the pain to where the function is called. So you modify those and it goes up the chain until you're at creation of the vector and you're left wondering how to make it have a static lifetime. Really, what the compiler means is that a reference isn't appropriate and you need to either move the data, copy the data, or use some other construct like the `Arc` (atomic reference counter) that is appropriate to the situation.

Memory that's kept around forever that is no longer useful is fundamentally very much like a memory leak, even if it is still possible to deallocate it in a hypothetical sense.

Holodeck Safeties are Offline

There's one last thing that we need, and it is dark and terrible magic. It is *unsafe*. Unsafe exists because it has to. The compiler would rather err on the side of caution and say no to a program that is correct, than say yes to one that isn't. You might also need to interact with some other library or do some very low-level stuff. For this reason, we can override this and tell the compiler that you promise you know this is okay. You do so at your own risk, though, because you can get it wrong and if you do you get all the same problems Rust tries to avoid ("undefined behaviour"), like race conditions, segmentation faults and memory leaks.

To do anything that qualifies as unsafe, you declare a block as `unsafe`. Inside an unsafe block, you can do the following things that you are not normally allowed to do [KNC20]:

1. Call an unsafe function/method

2. Access or modify a mutable static variable
3. Implement an unsafe trait
4. Access the fields of a `union`
5. Dereference a raw pointer

That list is probably less extensive than you were expecting. Declaring a block as unsafe does not grant you unlimited power, sadly. The borrow checker still does its thing and there are still rules.

Unsafe blocks are supposed to be small (this reduces the chance of an error and makes it easier to find) and their safety conditions should be well-documented...

There are also unsafe functions and traits, which do not grant superpowers, but instead impose additional restrictions. An unsafe function can only be called from inside an unsafe block. You specify that a given function is `unsafe` by putting that in the function signature; similarly, you say a trait is `unsafe` by putting `unsafe` before the trait declaration. But what's really going on is that an `unsafe` function is saying that its caller is responsible for ensuring some safety conditions, which it must document. For whatever reason, the Rust compiler cannot verify these safety conditions. Dually, if you implement a function from an unsafe trait, you have to ensure extra safety conditions. (Failing to do so when implementing the unsafe `Sync` trait has caused many unsound Rust bugs!)

Danger Zone. The easiest example to show is what happens when you want to call a function that is unsafe. Suppose we have a function `do_unsafe_thing()`; its function signature will be something like `unsafe fn do_unsafe_thing()` and to call it, we must wrap it in an unsafe block:

```
unsafe {
  do_unsafe_thing();
}
```

Anyone who wants to use an unsafe function also has to sign away their life, or at least their safety, by acknowledging that they know the function in question is unsafe (by enclosing the call in an unsafe block), and promising to ensure the necessary safety condtions. (Readbacks are a safety convention used in aviation, among other places.)

If you try to use an unsafe function without it being in an unsafe block, the compiler will, naturally, forbid such a thing. Just smashing the unsafe block around it is enough to make the compiler quiet, but not a thorough code reviewer. They would ask about whether you've read carefully the documentation of the function in question and whether you are sure you're calling it with the right arguments...You did read the documentation, right? Right?

Conversely, unsafe blocks don't have to be in unsafe functions; if not in an unsafe function, you're saying that the function is uncondtionally safe to call, i.e. you are encapsulating the unsafety.

Mutable static variables. Rust tries pretty hard to discourage you from using global variables, and they are right to do so. It's a quick shortcut and we do it a lot in course assignments, exercises, labs, and even exam questions. On an exam question, the thing I want to test is something like how you use the mutex and queue constructs to solve the problem, not how well you pass the mutex and queue pointers from the main thread to the newly created threads. In production code, though, global variables are really not recommended because of how harmful it is to good software engineering principles.

But anyway, you can make global variables mutable in Rust, if you must, and do so you have to mark this as unsafe. If you find yourself doing such a thing, please stop and think very carefully about why.

Unions. In C, there exists the concept of the `union`[11]. You might not have heard of it because a lot of people don't like it (and I'm one of them). You might have to contend with it in a particular API. It's like a `struct`, except where a `struct` is all of the contents (e.g., an integer and a floating point number and a pointer), a `union` is only one of those at a time (an integer or a floating point number or a pointer). Because there's no way to be totally sure that the `union` you're looking at is in fact the type you expect it to contain, you can only access the members in an unsafe block. ("Type punning" is what you can do when you are messing with the types in a union.)

[11] `https://en.wikipedia.org/wiki/Union_type`

Raw pointers. You can create raw pointers anywhere you like, but to dereference them, that has to be in an unsafe block. Creating the raw pointers can't cause a program crash; only using them does that. Of course, creating them incorrectly guarantees that when you try to use them they blow up in your face. I guess blame is a tricky subject.

Here's an example from the official docs [KNC20]:

```
let mut num = 5;
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1␣is:␣{}", *r1);
    println!("r2␣is:␣{}", *r2);
}
```

You can also use raw pointers when you need to write to a particular memory address, which sometimes happens for memory-mapped I/O. You just assign your value to an integer (i.e., `let add = 0xDEADBEEF`) and then cast it to a raw pointer (which is of type `*const`). To write some data to that address, use the unsafe block and write it.

You might need this if you are calling into a C library or function. The Rust universe of packages ("crates") is getting larger all the time, but sometimes you'll have to interact with a library in C...or write a part of your application in Rust that is called from C. There is a crate for cURL, but it might be interesting to learn what one would have to do to use the C library for it...

Rust vs C++. Now that you know all about Rust, you can see a discussion of the C++ equivalents here: `https://reberhardt.com/cs110l/spring-2020/slides/lecture-18.pdf#page=27`. Basically everything in Rust also exists in C++, but C++ doesn't make you use these things in a safe way.

Pitfalls

Over the years we've observed a few different things that people can get wrong about Rust and we'd like to just call them out before we close out the Rust topic. Hopefully it helps you avoid some errors. In no particular order:

- Overuse of `static` lifetime to band-aid ownership issues
- Using `clone()` much more than needed to try to avoid borrow-checker complexity
- Underestimating the overhead of array indexing rather than use of an iterator
- Unbuffered I/O
- Expensive operations like `resize()` on a vector
- Assuming everything will always go right and just using `unwrap()` with no handling of errors
- Vibe coding: the LLM's code may not be fully correct or optimal!

Use Libs Wisely!

In previous courses, such as a concurrency course, there was a lot of expectation to do most things the hard way: write your own implementation and don't use libraries. In this course, such restrictions don't apply. In industry, you'll use libraries that have appropriate functionality, assuming the license for them is acceptable to your project. It's strange that hiring interviews might ask you to implement a linked list, when in reality we would always expect that you would use one in a (standard) library...

Although the topic of using libraries used to have its own lecture, we've now moved that subject matter to the appendices for the course. It's worth taking a look at, though some examples relate to topics covered later in the course. Libraries like Crossbeam and Rayon are very useful and they are worth taking a look at – but not in the main lecture material. Use them as appropriate in the course examples, assignments, et cetera.

5 — Asynchronous I/O

Asynchronous/non-blocking I/O

To motivate the need for non-blocking I/O, consider some standard I/O code:

```
fn main() -> io::Result<()> {
  let mut file = File::open("hello.txt")?;
  let mut s = String::new();
  file.read_to_string(&mut s)?;
  Ok(())
}
```

(The ? operator "for easier error handling" is an alternative to `try!` and `unwrap`.)

This isn't very performant. The problem is that the `read` call will *block*. So, your program doesn't get to use the zillions of CPU cycles that are happening while the I/O operation is occurring.

As seen previously: threads. Threads can be fine if you have some other code running to do work—for instance, other threads do a good job mitigating the I/O latency, perhaps doing I/O themselves. But maybe you would rather not use threads. Why not?

- potential race conditions;
- overhead due to per-thread stacks; or
- limitations due to maximum numbers of threads.

Doing non-blocking I/O

We're going to focus on low-level I/O from sockets in this part of the lecture, using the `mio`[12] library from `tokio`. Async file I/O is also possible via `tokio::fs` and the ideas will carry over. One might often want to wrap the low-level I/O using higher-level abstractions, and the larger project `tokio.rs` is one way of doing that.

Fundamentally, there are two ways to find out whether I/O is ready to be queried: polling (under UNIX, implemented via `select`, `poll`, and `epoll`) and interrupts (under UNIX, signals). `mio` supports polling-based approaches and abstracts across Linux (via epoll), Windows (via IOCP), and BSDs including MacOS (via kqueue).

[12] `https://tokio-rs.github.io/mio/doc/mio/`

The key idea is to give `mio` a bunch of event sources and wait for events to happen. In particular:

- create a `Poll` instance;
- populate it with event sources e.g. `TCPListeners`; and,
- wait for events in an event loop (`Poll::poll()`).

Let's run through these steps in order, following `https://docs.rs/mio/0.7.0/mio/guide/index.html`:

Creating a `Poll` instance. Just use the API:

```
let poll = Poll::new()?;
let events = Events::with_capacity(128);
```

We're going to proactively create `events`; this data structure is used by `Poll::poll` to stash the relevant `Event` objects.

The `poll` object keeps track of event sources and, on request, pulls the events from the sources and puts them into the argument to `Poll::poll()`.

Populating the `Poll` instance. The docs refer to this as "registering event source". On all platforms this can be a socket (or lower-level networking source); on UNIX it can be also be a file descriptor.

```
let mut listener = TcpListener::bind(address)?;
const SERVER: Token = Token(0);
poll.registry().register(&mut listener, SERVER, Interest::READABLE)?;
```

The payload is the `register` call. Parameters, going right-to-left:

- You're telling it to check for when the `listener` indicates that something is available to read ("`READABLE`").
- The `SERVER` parameter is a note-to-self saying that events indicated with this particular `listener` should be flagged with the `SERVER` token. (Otherwise, if you register multiple listeners, you will have a bunch of events and not know which listener they came from.)
- Finally, the provided `listener` watches for connections on `address` (not provided here, but can be a `host:port` string).

Waiting on an `Poll` instance. Having completed the setup, we're ready to wait for events on any registered listener.

```
loop {
    poll.poll(&mut events, Some(Duration::from_millis(100)))?;

    for event in events.iter() {
        match event.token() {
            SERVER => loop {
                match listener.accept() {
                    Ok((connection, address)) => {
                        println!("Got a connection from: {}", address);
                    },
                    Err(ref err) if would_block(err) => break,
                    Err(err) => return Err(err),
                }
            }
        }
    }
}

fn would_block(err: &io::Error) -> bool {
    err.kind() == io::ErrorKind::WouldBlock
}
```

As foreshadowed, `poll.poll` will populate `events`, and waits for at most 100 milliseconds. A timeout of `None` will block until an event occurs.

Note the use of the `SERVER` token when processing the event. If there were multiple listeners, you would give them each a different token. Each event may correspond to one or more connections.

You can find the complete example here:

`https://docs.rs/mio/0.7.0/mio/struct.Poll.html`

Network Programming

If all you want to do is request a web page in Rust, use the `reqwest` library (`https://docs.rs/reqwest/0.10.8/reqwest/`), which has both blocking and non-blocking interfaces. Here's the non-blocking interface:

```
let body = reqwest::get("https://www.rust-lang.org")
    .await?
    .text()
    .await?;

println!("body = {:?}", body);
```

(If you are doing multiple requests, you should create your own `Client` and `get` from it instead of `reqwest::get`).

Back to the Futures. The use of `await` is a bit tricky. If you took CS 343 (for instance, because you are an SE student), then you will have seen the concept. Otherwise I'll briefly explain futures from first principles. You can find Rust documentation on them here:

`https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html`

The `get` function returns a *future*. What's that? It's an object that will, at some point in the future, return a second object.

Here's an analogy. I go to Ziggy's Cycles and try to purchase a bicycle. Since there's currently a pandemic going on in Canada as I write this in September 2020, and bicycles are more popular than usual, it's reasonable to expect that they might actually be out of bicycles at the moment, and so they can't give me a bicycle right away. But they'll take my money and specifications for a desired bicycle and give me a ticket (the future). Some time later, I can trade in that ticket (await) for an actual bicycle.

Plug-in Executors. There are many possible definitions of `async/await`, and the appropriate one depends on your context. Rust allows you to specify a runtime which defines the meaning of `async/await` for your program.

The simplest `await` just blocks and waits on the current thread for the result to be ready. A Rust library provides `futures::executor::block_on` with that simplest functionality.

```
use futures::executor::block_on;

async fn hello_world() {
  println!("hello");
}

fn main() {
  let future = hello_world();
  block_on(future);
}
```

Even that executor requires you to declare dependency `futures = "0.3"` in Cargo.toml; build with `cargo build` and run with `cargo run`. The full code is in the course repo under `live-coding/L05/block-on`.

`tokio` includes a more sophisticated executor as well; e.g. when there are multiple active `awaits`, `tokio` can multiplex them onto different threads. You can specify the `tokio` executor (or others) with a tag above `main()` and by declaring `main()` to be `async`, instead of what we did above with explicitly calling `block_on`. There are other tags to choose other executors (e.g. `async_std`).

```
#[tokio::main]
async fn main() {
    // do async stuff
}
```

You can read more about tokio here:

https://medium.com/@alistairisrael/demystifying-closures-futures-and-async-await-in-rust-part-3-async-await-9ed20eede7a4

Using libcurl: easy

libcurl is a C library for transferring files. It has Rust bindings and we'll explain how to use those.

First we'll start with the easy interface. This is a synchronous interface that uses callbacks. Here's some code from the Rust bindings documentation (`https://docs.rs/curl/0.4.33/curl/`):

```
use std::io::{stdout, Write};

use curl::easy::Easy;

// Write the contents of rust-lang.org to stdout
let mut easy = Easy::new();
easy.url("https://www.rust-lang.org/").unwrap();
easy.write_function(|data| { // callback function
    stdout().write_all(data).unwrap();
    Ok(data.len())
}).unwrap();
easy.perform().unwrap();
```

Note that we provide a lambda as a callback function. This lambda is to be invoked when the library receives data from the network (i.e. `write_function()`).

In the body of the lambda, we simply write the received data to `stdout` and return the number of bytes we processed (all of them, in this case). Looking at the original libcurl documentation (`https://curl.haxx.se/libcurl/c/CURLOPT_WRITEFUNCTION.html`), you'll see how the Rust bindings are a fairly straightforward translation.

We call `easy.perform()` to, well, perform the request, blocking until it finishes, and using the callback to process the received data.

Using libcurl: multi

The real reason we're talking about libcurl is the asynchronous multi interface; network communication is a great example of asynchronous I/O. You can start a network request and move on to creating more without waiting for the results of the first one. For requests to different recipients, it certainly makes sense to do this.

The main tool here is the "multi handle"—this is a structure that lets us have more than one curl easy handle. And rather than waiting, we can start them and then check on their progress.

The structure for the new multi-handle type is `curl::multi::Multi` (instead of `curl::easy::Easy`) and it is initialized with the `new()` function. The multi functions may return a `MultiError` rather than the easy `Error`, and I don't know how to unify the error handling with ? here.

Once we have a multi handle, we can add easy objects—however many we need—to the multi handle. Creation of the easy object is the same as it is when being used alone—use `Easy::new()` to create it and set options on that handle. The documentation suggests that an `Easy2` might be better for use with multi handles. Then, we add the easy (or easy2) object to the multi handle with `add()` (or `add2()`). The `add()` or `add2()` functions return an actual easy handle.

Once we have finished putting all the easy handles into the multi handle, we can dispatch them all at once with `perform()`. This function returns, on success, the number of easy handles in that multi handle that are still running. If it's down to 0, then we know that they are all done. If it's nonzero it means that some of them are still in progress.

This does mean that we're going to call `perform()` more than once. Doing so doesn't restart or interfere with anything that was already in progress—it just gives us an update on the status of what's going on. We can check as often as we'd like, but the intention is of course to do something useful while the asynchronous I/O request(s) are going on. Otherwise, why not make it synchronous?

Suppose we've run out of things to do though. What then? Well, we can wait, if we want, using `wait()`. This function will block the current thread until something happens (some event occurs).

The first parameter to `wait()` is an array of extra file descriptors you can wait on (but we will always want this to be `&mut []` in this course). The second parameter is a maximum time to wait. The return value is the actual number of "interesting" events that occurred (interesting is the word used in the specifications, and what it means is mysterious). For a simple use case you can ignore most of the parameters and just wait for something to happen and go from there.

In the meantime though, the perform operations are happening, and so are whatever callbacks we have set up (if any). And as the I/O operation moves through its life cycle, the state of the easy handle is updated appropriately. Each easy handle has an associated status message as well as a return code.

Why both? Well—one is about what the status of the request is. The message could be, for example, "done", but does that mean finished with success or finished with an error? The second one tells us about that. We can allegedly ask about the status of the request using `messages()`, and we're really supposed to do that if we use `action()` (which we don't talk about) rather than `perform()` (which we do).

We pass `messages()` a callback which finds out what happened and makes sure all is well. This callback is an `FnMut` and hence allowed to mutate state. What we are looking for is that the callback's parameter `msg` has `result_for` including `Some`—request completed. If not, this request is still in progress and we aren't ready to evaluate whether it was successful or not. If there are more handles to look at, we should go on to the next. If it is done, we should look at the result. If it is `Error` then there is an error. Else, everything succeeded.

When a handle has finished, you need to remove it from the multi handle. Remove the handle you got back from `add/2` with `remove/2`. You don't have to cleanup the easy handle because Rust.

Let's consider the following code example by Clemens Gruber [Gru13], which I've translated to Rust (mostly). This example puts together all the things we talked about in one compact code segment. Here, the callback prints the data to `stdout`.

```
const URLS:[&str; 4] = [
  "https://www.microsoft.com",
  "https://www.yahoo.com",
  "https://www.wikipedia.org",
  "https://slashdot.org" ];

use curl::Error;
use curl::easy::{Easy2, Handler, WriteError};
use curl::multi::{Easy2Handle, Multi};
use std::time::Duration;
use std::io::{stdout, Write};

struct Collector(Vec<u8>);
impl Handler for Collector {
    fn write(&mut self, data: &[u8]) -> Result<usize, WriteError> {
        self.0.extend_from_slice(data);
        stdout().write_all(data).unwrap();
        Ok(data.len())
    }
}

fn init(multi:&Multi, url:&str) -> Result<Easy2Handle<Collector>, Error> {
    let mut easy = Easy2::new(Collector(Vec::new()));
    easy.url(url)?;
    easy.verbose(false)?;
    Ok(multi.add2(easy).unwrap())
}

fn main() {
    let mut easys : Vec<Easy2Handle<Collector>> = Vec::new();
    let mut multi = Multi::new();
```

```
    multi.pipelining(true, true).unwrap();
    for u in URLS.iter() {
        easys.push(init(&multi, u).unwrap());
    }
    while multi.perform().unwrap() > 0 {
        // .messages() may have info for us here...
        multi.wait(&mut [], Duration::from_secs(30)).unwrap();
    }

    for eh in easys.drain(..) {
        let mut handler_after:Easy2<Collector> = multi.remove2(eh).unwrap();
        println!("got response code {}", handler_after.response_code().unwrap());
    }
}
```

You may wonder about re-using an easy handle rather than removing and destroying it and making a new one. The official docs say that you can re-use one, but you have to remove it from the multi handle and then re-add it, presumably after having changed anything that you want to change about that handle.

Because a handle could be replaced with another one (or the same one), you could have a situation where there are constantly handles in progress and you might never be at a situation where there are no messages left. And that is okay.

In this example all requests had the same callback, but of course you could have different callbacks for different easy handles if you wanted them to do different things.

How well does this scale? The developer claims that you can have multiple thousands of connections in a single multi handle[13]. And 60k ought to be enough for anyone!

I enjoy pain! You can use cURL with `select()` if you wish, although it comes with an anti-recommendation: I think you shouldn't do it. But you can if you want. In some ways, cURL does make things less painful because it does some of the grunt work for you. Don't do it. Please no.

Building Servers: Concurrent Socket I/O

Your Choices. The first two both use blocking I/O, while the second two use non-blocking I/O [Lov13]:

- Blocking I/O; 1 process per request.
- Blocking I/O; 1 thread per request.
- Asynchronous I/O, pool of threads, callbacks, each thread handles multiple connections.
- Nonblocking I/O, pool of threads, multiplexed with select/poll, event-driven, each thread handles multiple connections.

Blocking I/O; 1 process per request. This is the old Apache model.

- The main thread waits for connections.
- Upon connect, the main thread forks off a new process, which completely handles the connection.
- Each I/O request is blocking, e.g., reads wait until more data arrives.

Advantage:

- "Simple to understand and easy to program."

Disadvantage:

- High overhead from starting 1000s of processes. (We can somewhat mitigate this using process pools).

This method can handle ∼10 000 processes, but doesn't generally scale beyond that, and uses many more resources than the alternatives.

[13] See this post from the mailing list: `https://curl.haxx.se/mail/lib-2011-11/0078.html`

Blocking I/O; 1 thread per request. We know that threads are more lightweight than processes. So let's use threads instead of processes. Otherwise, this is the same as 1 process per request, but with less overhead. I/O is the same—it is still blocking.

Advantage:

- Still simple to understand and easy to program.

Disadvantages:

- Overhead still piles up, although less than processes.
- New complication: race conditions on shared data.

Asynchronous I/O. The other two choices don't assign one thread or process per connection, but instead multiplex the threads to connections. We'll first talk about using asynchronous I/O with select or poll.

Here are (from 2006) some performance benefits of using asynchronous I/O on lighttpd [Tea06].

version		fetches/sec	bytes/sec	CPU idle
1.4.13	sendfile	36.45	3.73e+06	16.43%
1.5.0	sendfile	40.51	4.14e+06	12.77%
1.5.0	linux-aio-sendfile	72.70	7.44e+06	46.11%

(Workload: 2×7200 RPM in RAID1, 1GB RAM, transferring 10GBytes on a 100MBit network).

The basic workflow is as follows:

1. enqueue a request;
2. ... do something else;
3. (if needed) periodically check whether request is done; and
4. read the return value.

See the ECE 252 notes if you want to learn about select/poll!

6 — Modern Processors

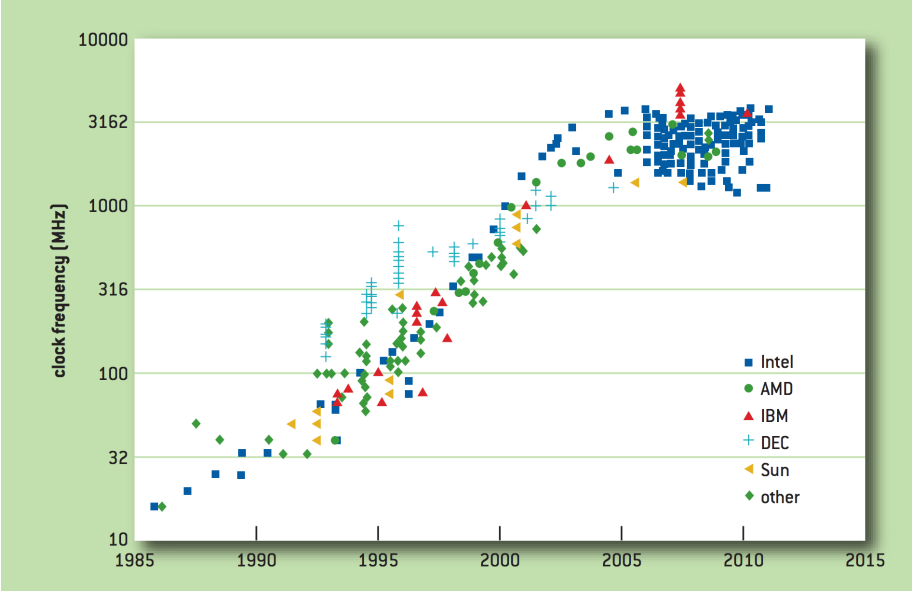
You know how `http://computers-are-fast.github.io/` featured in Lecture 1? It may also feature on your exams. You might want to print out your results and bring them.

Modern Processors

It's critical to understand what's going on with the hardware if we want to write good programs. This lecture is based off the talk by Cliff Click [CG10].

Remember the classic von Neumann machine architecture. A program is comprised of both instructions and data, both of which are stored in the same memory. A program executes sequentially, one statement at a time, one after another. That is not really how computers work, at least not anymore, but it is an abstraction we still maybe find useful when it comes to algorithm analysis.

Consider this graph of CPU clock speed (frequency) over time from [DKM+12]:



Clearly there is an area in which frequency scaling was effective. Next year's CPU would have a higher clock speed, and higher clock speed means more cycles per second, and more cycles per second means more work is done in a given second, and that means better performance. Except, we hit the wall: clock speeds stop getting faster around 2005, stopping at around 3 GHz. Speeding them up beyond this would take, well, more voltage which means more power and more heat, and more heat means higher failure/error rates, and more cooling, and the cooling takes power too, and all that waste heat, well, it will eventually, at the end of this chain, make polar bears sad.

Digression: if we look at the x86 processor, one with which everyone is probably at least *passingly* familiar, it is a Complex Instruction Set Computer (CISC) processor. In other words, there are a lot of assembly instructions. But why? This was intended for your convenience as a programmer: if you were going to write assembly, wouldn't it be

nice to have a sine function that takes one argument instead of having to grind out (or copy-paste) the calculation of a sine routine every single time you needed it? So the hardware people thought they were doing everyone a favour. These are easy to program in, from the way the assembly programmer thinks, but hard to implement and hard to pipeline.

For a lot of CISC machines, the Cycles Per Instruction (CPI) varied, something like 4-10 cycles to complete any instruction, but at least it was predictable. Every time, no matter what, it takes the same number of cycles. Program performance was basically the number of page faults (disk accesses) times the amount of time it takes to read from disk, plus the instruction execution time (which is generally small compared to page fault service times)[14]. Thus the optimization goal is: minimize page faults. Page fault count is relatively easy to measure and there are some things we can do to reduce the number of page faults: optimize our data access patterns, change how we pack the data, et cetera. If you were working with an embedded system with no disk (or at least no page faults) then the optimization goal is minimize instruction count.

Between 1990 and 2005 we got some really impressive scaling on CPU frequency. This was caused by a few factors. The first is the advent of the Reduced Instruction Set Computer (RISC) CPU: simpler processors are easier to scale than complex ones, and simpler instructions mean fewer cycles per instruction. That also means we can have more pipelining. The tradeoff is that RISC CPUs are much harder to program in assembly directly, so compilers had to do the work. The example in [CG10] is delay slots: an instruction after a branch is always executed or, worse, the result of a computation is not available to the next instruction. In these cases the "simple" solution is to put a `NOP` (do-nothing) instruction in. But good compilers (and programmers) can rearrange instructions, hopefully, to make this work without wasting time. And another thing: memory got cheaper, so we have more of it, so page faults occurred less and less frequently and that's really something.

But then, as we have seen, we hit the (power) wall. And you might think, well, if I run into a wall, I can just go around it. There must be other ways to advance! And there are, except, we hit three other walls too, and now we are surrounded[15]. What are these other three seemingly-insurmountable barriers?

The first is instruction level parallelism (ILP) is getting close to the limit of what we can do. We can predict branches with a certain accuracy but if we have already got 95% efficiency, no matter how much time and effort and money is invested into improving the branch prediction routine we get maximally a 5% increase in branch prediction accuracy which translates into a very small speedup to the execution when we consider just how often a misprediction is the cause of the problem (5% of 5% is very small…just making up numbers).

The speed of memory advances has not at all kept up with the advances in CPU technology, so now we have moved from the era of runtime being dominated by page faults to the era of runtime being dominated by cache misses. Adding more cache isn't a perfect solution though, and doubling, say, level one cache (at great expense) does not double the speed of the program; it may speed it up by a small amount at most.

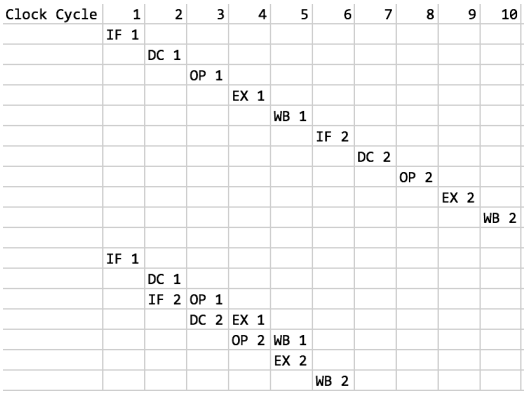
The final wall is the universal speed limit: the speed of light (curse you Einstein!). The more complex the CPU is, the longer the path any signal may have to travel to get from A to B. This is limited, most practically, by the speed of light, and thus far, nobody has invented a way to get around this universal speed limit (but we are working on it, and according to Star Trek, should have this sorted out by 2063 or so).

But let's go back to the subject of ILP. Branch prediction and pipelining have been touched upon but there is so much more to it. The idea with ILP is not having more cycles to work with, but instead, doing more in each clock cycle. And there's a lot of clever ideas.

Pipelining: you may have heard a bit about this already, especially so if you have taken a CPU architecture course. To complete an instruction there are five basic steps: (1) fetch the instruction from memory, (2) decode the instruction, (3) fetch needed operands, (4) perform the operation, and (5) write the result. So to do an instruction like `ADD R1, R2`, we need to fetch the instruction, decode it and figure out what is to be done, read the values from `R1` and `R2`, do the addition, and then write the result to `R1`. Thus even a simple instruction takes more than one clock cycle, but the good news is that the stages can overlap:

[14] For further discussion about this, see the ECE 254 notes about page faults and caching and disk read times.

[15] "He is intelligent, but not experienced. His pattern indicates two dimensional thinking." - Spock, *Star Trek II: The Wrath of Khan*



In the above image, two instructions are shown. The top part shows no pipelining; the bottom shows what happens when pipelining is used. Each part of the instruction must be done sequentially—the instruction cannot be decoded until it is fetched—but at least the next instruction can be done. So it allows each of these to appear as if it is 1 clock cycle. If all goes well, then you complete one instruction per clock cycle.

But there are pipeline hazards: sometimes you need the result of a previous step before you can go on. These prevent us from reaching the theoretical maximum of one instruction completed per clock cycle. Needing a previous result is not the only kind of hazard, though; we may have conflicts over certain CPU resources (how many floating point units are there, after all...?) or, fetch may be unable to identify the next instruction because of a branch. In the worst case, if we have mispredicted a branch, we have to flush the pipeline: throw away the instructions fetched, decoded, operands prepared, et cetera, because we guessed wrong and started doing the wrong actions. In that case, some extra work was done that was not necessary...

The next idea relates to getting items from memory. If we do a load from memory, and we are lucky, it is found in the fastest cache and is available in perhaps 2-3 clock cycles. If we must go to memory, it may be 200-300 cycles. If it is in level 2 or level 3 cache it will be somewhere in between. The key idea, though, is if we are trying to load something into register `R7`, it will take time until that value is actually present in that register. The simplest approach is to just wait around until that value has arrived, and then go on to the next instruction. That works, but we can do better.

That better idea is: continue executing until `R7` is used somewhere. This allows us to get some useful work done in the meantime. Hardware keeps track of the fact that the register is not quite ready yet, but the work can get done in what is called the "miss shadow". It's possible to have more than one load in flight at a time. Two or more can be done in various CPU architectures, but it is of course hardware dependant.

Branch prediction has come up already, but if we have a load followed by a compare used as a branch, we can then, well, guess. If we are wrong, there is the need to cleanup. But the good news is that branch prediction is usually right most of the time, perhaps 95% or more (we'll definitely return to this later).

Another nice thing that the hardware can do for us is "dual issue" instructions. If we have two consecutive instructions that both take the same amount of time, use unrelated registers, and don't consume two of the same resource, we can start both instructions at once. If the instructions are `ADD R1, 16` and `CMP R2, 0` they do different things with different registers so there is no reason these cannot be done in parallel (if there are enough fetch/decode/etc units). In an embedded system, you may be interested in ensuring that this happens during a computationally intensive loop, such as encoding/decoding of media. If programmed correctly, you can be sure you get dual issue on every cycle.

Then a group of things that somewhat go together: register renaming, branch prediction, speculation, and Out-of-Order (O-O-O) Execution. These all work synergistically: each adds to the benefits the other brings. Register renaming works on a fairly simple principle: an assembly instruction says to read from register `R4`, but behind the scenes inside the processor, it is mapped to a physical register (let's say `RA` for the purpose of the example). Consider the following assembly instructions:

```
MOV R2, R7 + 32
ADD R1, R2
MOV R2, R9 + 64
```

```
ADD R3, R2
```

Under normal circumstances, we cannot do instruction 3 until instruction 2 has been completed because we need the value of R2 that was put in there (taken from memory somewhere) to be added to R1. Except, with register renaming, behind the scenes the first two instructions may replace R2 with RX and the second pair of instructions have R2 replaced with RY and these things can take place in parallel, or without a stall, at the very least.

This has a certain synergy with branch prediction. If we predict a branch, we can do speculative changes into one set of registers while we keep the "old" register values around too. When we figure out whether the branch prediction is correct, we can then get rid of the ones we don't need: the originals if predicted correctly, and the new values otherwise. So we get better recovery if there is a misprediction. Actually, I bet students wish they could do this: write down both answers to a question and let the TA pick the correct one at the end...

Most importantly, it allows us to get past a cache miss and keep going; the goal here is to run until we can start the next cache miss, because the sooner that starts the sooner it's over, and the faster the program executes, ultimately. A quick example from the presentation demonstrates this, in x86 assembly [CG10]:

```
ld rax, rbx+16   ; assume cache miss
add rbx, 16      ; carry on anyway, ADD doesn't need rax value from LD
                 ; register renaming => LD (write to reg)/ADD (read from reg) don't interfere
cmp rax, 0       ; needs rax value, queue till available
jeq null_chk     ; oops! need cmp result
                 ; speculate: assume branch not taken
st rbx-16, rcx   ; speculatively store to store buf (not L1)
ld rcx, rdx      ; unrelated cache miss: 2 misses now active, 1 speculative
ld rax, rax+8    ; now must wait for result of first LD
```

To summarize: there are seven operations we were trying to do here with two cache misses. The cache misses complete in cycles 300 and 304 (maybe 302 if we have dual issue), so in total we complete 7 operations in about 305 cycles. All the trickery and cleverness got us to that second miss which means we complete in 305. If we did not manage that, it would take about 600 cycles to complete it all. So we did double performance, even though in this example our overall performance was terrible.

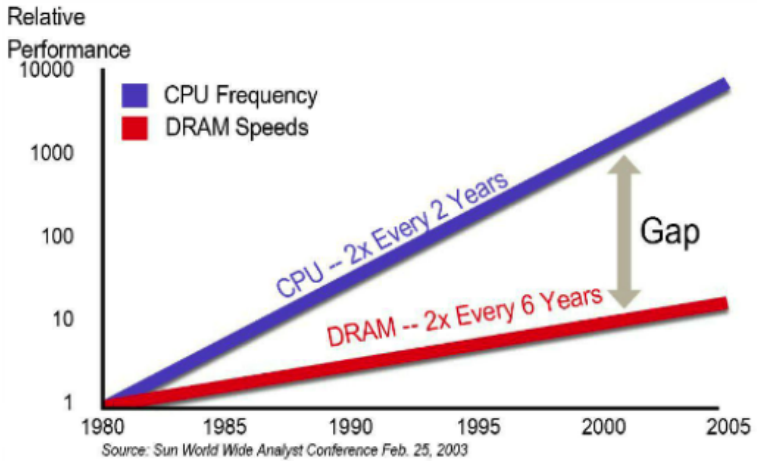
For years Intel was trying to push its Itanium processors (which were so unsuccessful they got the nickname "Itanic". Ouch). The goal of these was to find static (compile-time) parallelism: if a machine has infinite registers, can speculate infinitely, etc, the program gets sped up. Run all possibilities in parallel and at the end figure out which is right (wasn't this a Nicolas Cage movie?). Unfortunately it didn't work out very well because this requires the right kind of program and a super smart compiler. Oh yes, and infinite registers requires infinite space as well as infinite money. So instead the quest has turned to how we can get better performance out of x86...

The x86 approach tries to maximize dynamic (run-time) parallelism. This has been done incrementally, with more pipelining, re-order buffers, adding more functional units, and so on. But the walls are still there: cache miss rates and branch mispredicts continue to dominate performance, even though the rates are very low, because a miss costs so much.

How are we doing so far? Well, here's a short video that goes over where we were in the beginning of 2017: `https://www.youtube.com/watch?v=4A0Iks4ENIs` with the launch of Intel's latest (at the time) processor. But since then, AMD launched Ryzen—Never turn your back on Threadripper!—and it's forced Intel to compete again... In 2020 the situation is more like `https://www.youtube.com/watch?v=a8apEJ5Zt2s`.

According to [CG10] something like 90-99% of the transistors on a modern x86 chip are spent in cache. In spite of the extreme complexity of the decode logic that allows multiple parallel decodes of all the weird and wacky instructions of the x86, pushing cache to the biggest size it can be is so important because it prevents the performance hit of going to memory.

The image below (from Sun World Wide Analyst Conference in 2003) is obviously a bit dated but this is very instructive as to the trend:



DRAM is, however, not the only kind of memory. There is SRAM (Static RAM) which is fast but expensive, the kind of stuff that goes on the CPU die, and it is six transistors per bit. Compare against DRAM which is much cheaper, but slow: one transistor and one capacitor per bit. Improvements in DRAM have not really improved latency but have improved bandwidth; DDR (Dual Data Rate... not Dance Dance Revolution) means there are two transfers per cycle, but it still takes significant time to get any data out. And DRAM needs occasional refreshes (capacitors...) so sometimes we have to wait for that.

In the Operating Systems course you probably learned that disk is the slowest thing and the limiting factor. That's true, as Obi-Wan Kenobi would say, from a certain point of view. Now that we live in the world of Solid State Drives (SSDs), "disk" reads are about as fast as memory reads and memory reads are the rate-limiting step in the system. Nonvolatile memory looks to be even faster. More is the new more, orange is the new black, and memory is the new disk.

To get memory access speed up there are things we can do, like relax coherency constraints, more synchronization through locks... all of which we will come back to in some upcoming lectures.

If we want to get better performance, we need to figure out where time is going. For that we will have the subject of profiling, which comes up in some later lectures. If we can track down where our cache misses are occurring, maybe, just maybe, we can do something about it.

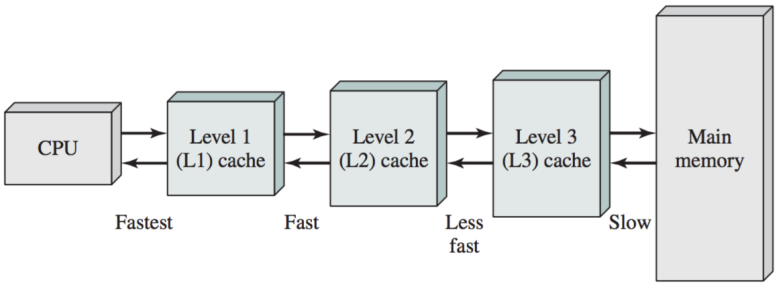
A Deeper Look at Cache Misses

As discussed, the CPU generates a memory address for a read or write operation. The address will be mapped to a page. Ideally, the page is found in the cache, because that would be faster. If the requested page is, in fact, in the cache, we call that a cache *hit*. If the page is not found in the cache, it is considered a cache *miss*. In case of a miss, we must load the page from memory, a comparatively slow operation. The percentage of the time that a page is found in the cache is called the *hit ratio*, because it is how often we have a cache hit. We can calculate the effective access time if we have a good estimate of the hit ratio (which is not overly difficult to obtain) and some measurements of how long it takes to load data from the cache and how long from memory. The effective access time is therefore computed as:

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m,$$

where h is the hit ratio, t_c is the time required to load a page from cache, and t_m is the time to load a page from memory. Of course, we would like the hit ratio to be as high as possible.

Caches have limited size, because faster caches are more expensive. With infinite money we might put everything in registers, but that is rather unrealistic. Caches for memory are very often multileveled; Intel 64-bit CPUs tend to have L1, L2, and L3 caches. L1 is the smallest and L3 is the largest. Obviously, the effective access time formula needs to be updated and expanded when we have multiple levels of cache with different access times and hit rates. See the diagram below:



Three levels of cache between the CPU and main memory [Sta14].

If we have a miss in the L1 cache, the L2 cache is checked. If the L2 cache contains the desired page, it will be copied to the L1 cache and sent to the CPU. If it is not in L2, then L3 is checked. If it is not there either, it is in main memory and will be retrieved from there and copied to the in-between levels on its way to the CPU.

Cliff Click said that 5% miss rates dominate performance. Let's look at why. I looked up a characterization of the SPEC CPU2000 and CPU2006 benchmarks [KVN+08].

Here are the reported cache miss rates[16] for SPEC CPU2006.

L1D	40‰
L2	4 ‰

Let's assume that the L1D cache miss penalty is 5 cycles and the L2 miss penalty is 300 cycles, as in the video. Then, for every instruction, you would expect a running time of, on average:

$$1 + 0.04 \times 5 + 0.004 \times 300 = 2.4.$$

Misses are expensive!

If we replace the terms t_c and t_m with t_m and t_d (time to retrieve it from disk) respectively, and redefine h as p, the chance that a page is in memory, we can get an idea of the effective access time in virtual memory:

$$\text{Effective Access Time} = p \times t_m + (1 - p) \times t_d.$$

And just while we're at it, we can combine the caching and disk read formulæ to get the true effective access time for a system where there is only one level of cache:

$$\text{Effective Access Time} = h \times t_c + (1 - h)(p \times t_m + (1 - p) \times t_d).$$

This is good, but what is t_d? This is a measurable quantity so it is possible, of course, to just measure it[17].

The slow step in all of this, is obviously, the amount of time it takes to load the page from disk. According to [SGG13], restarting the process and managing memory and such take something like 1 to 100 μs. A typical hard drive in their example has a latency of 3 ms, seek time (moving the read head of the disk to the location of the page) is around 5 ms, and a transfer time of 0.05 ms. So the latency plus seek time is the limiting component, and it's several orders of magnitude larger than any of the other costs in the system. And this is for servicing a request; don't forget that several requests may be queued, making the time even longer.

Thus the disk read term t_d dominates the effective access time equation. If memory access takes 200 ns and a disk read 8 ms, we can roughly estimate the access time in nanoseconds as $(1 - p) \times 8\ 000\ 000$.

If the page fault rate is high, performance is awful. If performance of the computer is to be reasonable, the page fault rate has to be very, very low.

[16] ‰ is "permil", or per-1000.

[17] One of my favourite engineering sayings is "Don't guess; measure." You may be sick of hearing me say that one by now.

Summary: misses are not just expensive, they hurt performance more than anything else.

7 — CPU Hardware, Branch Prediction

Multicore Processors

As I've alluded to earlier, multicore processors came about because clock speeds just aren't going up anymore. We'll discuss technical details today. Each processor *core* executes instructions; a processor with more than one core can therefore simultaneously execute multiple (unrelated) instructions.

Speed and Heat. The whole reason we're here with multicores is that it's impossible to continue to crank up clock rates, because the chips get too hot. Back in the 2000s, before they figured to put thermal sensors, I would occasionally hear about CPUs catching fire. These days, Intel and AMD chips support burst speeds which are faster than normal clock rates, and which can be used until the chips get too hot. So the base rate hasn't gotten much above 3GHz in recent decades, but burst rates are higher.

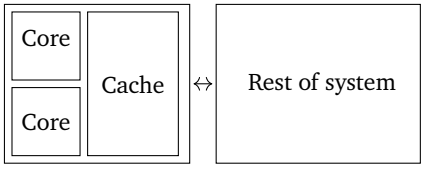
Chips and cores. Multiprocessor (usually SMP, or symmetric multiprocessor) systems have been around for a while. Such systems contain more than one CPU. We can count the number of CPUs by physically looking at the board; each CPU is a discrete physical thing.

Cores, on the other hand, are harder to count. In fact, they look just like distinct CPUs to the operating system:

```
plam@plym:~/courses/p4p/lectures$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model  : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
...
processor : 1
vendor_id : GenuineIntel
cpu family : 6
model  : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
```

If you actually opened my computer, though, you'd only find one chip. The chip is pretending to have two *virtual CPUs*, and the operating system can schedule work on each of these CPUs. In general, you can't look at the chip and figure out how many cores it contains.

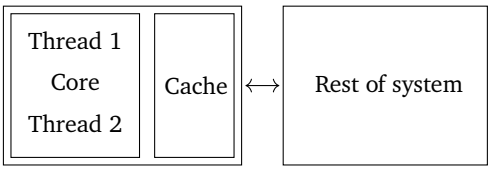
Hardware Designs for Multicores. In terms of the hardware design, cores might share a cache, as in this picture:



(credit: *Multicore Application Programming*, p. 5)

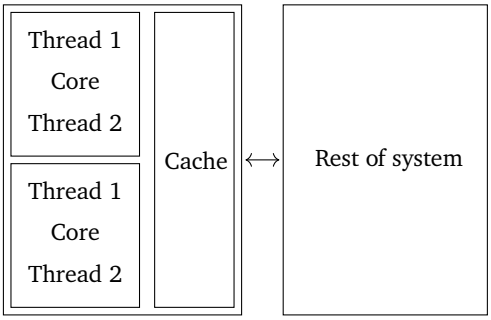
This above Symmetric Multithreading (SMP) design is especially good for the 1:1 threading model. In this case, the design of the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

Or, we can have a design that works well for the N:1 model (typically called hyperthreading):



One would expect that executing two threads on one core might mean that each thread would run more slowly. It depends on the instruction mix. If the threads are trying to access the same resource (e.g. Arithmetic Logic Units), then each thread would run more slowly. If they're doing different things, there's potential for speedup.

Finally, one can both use many cores and many threads on one core, as in the M:N model (also hyperthreading):



Here we have four hardware threads; pairs of threads share hardware resources. Looking at the specs of the AMD Ryzen 5 7600 (Zen 4), I see that it has 6 cores and 12 threads. Everyone does this nowadays. On the Zen 4 chips, each core has its own L1 and L2 caches, while the L3 caches are shared among the cores on the same so-called "Core Complex Die".

Non-SMP systems. The designs we've seen above have been more or less SMP designs; all of the cores are mostly alike. A very non-SMP system is the Cell, which contains a PowerPC main core (the PPE) and 7 Synergistic Processing Elements (SPEs), which are small vector computers. Recent Intel and AMD processors do have cores that provide the same programming interface but have different performance characteristics. Intel chips specifically have P (performance) and E (efficient) cores. Thinking back to bandwidth vs latency, E cores are higher-latency, but they're smaller and so Intel can put more of them on the chip. P cores are good for low-latency. AMD has c-cores which seem to do the same thing as Intel's E cores.

Non-Uniform Memory Access. In SMP systems, all CPUs have approximately the same access time for resources (subject to cache misses). There are also NUMA, or Non-Uniform Memory Access, systems out there. In that case, CPUs can access different resources at different speeds. (Resources goes beyond just memory).

In this case, the operating system should schedule tasks on CPUs which can access resources faster. Since memory is commonly the bottleneck, each CPU has its own memory bank.

Using CMT effectively. Typically, a CPU will expose its hardware threads using virtual CPUs. It is also the responsibility of the scheduler to make sure that tasks are scheduled on the best cores/virtual CPUs.

However, performance varies depending on context. In the above example, two threads running on the same core will most probably run more slowly than two threads running on separate cores, since they'd contend for the same core's resources. Task switches between cores (or CPUs!) are also slow, as they may involve reloading caches.

Solaris "processor sets" enable the operating system to assign processes to specific virtual CPUs, while Linux's "affinity" keeps a process running on the same virtual CPU. Both of these features reduce the number of task switches, and processor sets can help reduce resource contention, along with Solaris's locality groups.[18]

[18] Gove suggests that locality groups help reduce contention for core resources, but they seem to help more with memory.

Branch Prediction and Misprediction

The compiler (and the CPU) take a look at code that results in branch instructions such as loops, conditionals, or the dreaded `goto`[19], and it will take an assessment of what it thinks is likely to happen. By default I think it's assumed that backward branches are taken and forward branches are not taken (but that may be wrong). Well, how did we get here anyway?

In the beginning the CPUs and compilers didn't really think about this sort of thing, they would just come across instructions one at a time and do them and that was that. If one of them required a branch, it was no real issue. Then we had pipelining: the CPU would fetch the next instruction while decoding the previous one, and while executing the instruction before. That means if evaluation of an instruction results in a branch, we might go somewhere else and therefore throw away the contents of the pipeline. Thus we'd have wasted some time and effort. If the pipeline is short, this is not very expensive. But pipelines keep getting longer...

So then we got to the subject of branch prediction. The compiler and CPU look at instructions on their way to be executed and analyze whether it thinks it's likely the branch is taken. This can be based on several things, including the recent execution history. If we guess correctly, this is great, because it minimizes the cost of the branch. If we guess wrong, we have to flush the pipeline and take the performance penalty.

The compiler and CPU's branch prediction routines are pretty smart. Trying to outsmart them isn't necessarily a good idea. It's possible to give `gcc` some hints: we say either something is likely or unlikely.

These hints tell the compiler some information about how it should predict. It will then arrange the instructions in such a way that, if the prediction is right, the instructions in the pipeline will be executed. But if we're wrong, then the instructions will have to be flushed.

From what I can tell, the core Rust team isn't super comfortable with the idea of exposing these kinds of internal-compiler things, but there is an implementation of the likely/unlikely concept. You can sort of use it, but it could break in the future, as an experimental feature. If you want the experimental features enabled, you have to be using nightly build of Rust and to specify the feature at the top of your source file (e.g., `#![feature(core_intrinsics)]`)

Do they work? Here's a sample program to find out. I'll first test it with no hint, then putting `likely()` around the if condition, and then `unlikely()`, and show you the results.

```
fn f(a: i32) -> i32 {
    a
}

fn main() {
    let size = 100000;
    let large_vector = vec![0; size];
    let mut m1 = 0;
    let mut m2 = 0;

    for _j in 0..1000 {
        for k in 0..size {
            if *large_vector.get(k).unwrap() == 0 {
                m1 = f(m1 + 1)
            } else {
                m2 = f(m2 + 1)
            }
        }
    }
    println!("m1 = {}; m2 = {}", m1, m2);
}
```

And the results:

```
No hint at all:
  Time (mean +/- ?):      6.657 s +/-  0.144 s    [User: 6.614 s, System: 0.029 s]
  Range (min ... max):    6.413 s ...  6.905 s    10 runs
```

[19] Which I still maintain is a swear word in C.

```
Likely:
  Time (mean +/- ?):      6.762 s +/-  0.175 s    [User: 6.729 s, System: 0.028 s]
  Range (min ... max):    6.590 s ...  7.200 s    10 runs

Unlikely:
  Time (mean +/- ?):      6.943 s +/-  0.200 s    [User: 6.893 s, System: 0.033 s]
  Range (min ... max):    6.732 s ...  7.309 s    10 runs
```

Looks like hints don't help very much in this program at all. They made it marginally worse, not better. And getting it wrong comes with a penalty, too. This program might not be the ideal test case for hints, in that there might be a different scenario where the hints have a positive impact. However, we have at least established that hints aren't always a benefit, even if we know we're right. Under a lot of circumstances then, it's probably best just to leave it alone, unless we're really, really, really sure.

Conclusion: it's hard to outsmart the compiler. Maybe it's better not to try.

How does branch prediction work, anyway?

We can write software. The hardware will make it fast. If we understand the hardware, we can understand when it has trouble making our software fast.

You've seen how branch prediction works in ECE 222. However, we'll talk about it today in the context of performance. Notes based on a transcript of a talk by Dan Luu [Luu17].

I want you to pick up two points from this discussion:

- how branch predictors work—this helps you understand some of the apparent randomness in your execution times, and possibly helps you make your code more predictable; and,
- applying a (straightforward) expected value computation to predict performance.

Let's consider the following assembly code:

```
branch_if_not_equal x, 0, else_label
// Do stuff
goto end_label
else_label:
// Do things
end_label:
// whatever happens later
```

The `branch` instruction may be followed by either "stuff" or "things". The pipeline needs to know what the next instruction is, for instance to fetch it. But it can't know the next instruction until it almost finishes executing the branch. Let's look at some pictures, assuming a 2-stage pipeline.

With no prediction, we need to serialize:

bne.1	bne.2		
		things.1	things.2

Let's predict that "things" gets taken. If our prediction is correct, we save time.

bne.1	bne.2	
	things.1	things.2

But we might be wrong and need to throw out the bad prediction.

bne.1	bne.2		
	~~things.1~~		
		stuff.1	stuff.2

Cartoon model. We need to quantify the performance. For the purpose of this lecture, let's pretend that our pipelined CPU executes, on average, one instruction per clock; mispredicted branches cost 20 cycles, while correctly-predicted branches cost 1 cycle. We'll also assume that the instruction mix contains 80% non-branches and 20% branches. So we can predict average cycles per instruction.

With no prediction (or always-wrong prediction):

$$\text{non\_branch\_\%} \times 1 \text{ cycle} + \text{branch\_\%} \times 20 \text{ cycles} = 4.8 \text{ cycles}.$$

With perfect branch prediction:

$$\text{non\_branch\_\%} \times 1 \text{ cycle} + \text{branch\_\%} \times 1 \text{ cycle} = 1 \text{ cycle}.$$

So we can make our code run 4.8× faster with branch prediction!

Predict taken. What's the simplest possible thing? We can predict that a branch is always taken. (Loop branches, for instance, account for many of the branches in an execution, and are often taken.) If we got 70% accuracy, then our cycles per instruction would be:

$$(0.8 + 0.7 \times 0.2) \times 1 \text{ cycle} + (0.3 \times 0.2) \times 20 \text{ cycles} = 2.14 \text{ cycles}.$$

The simplest possible thing already greatly improves the CPU's average throughput.

Backwards taken, forwards not taken (BTFNT). Let's leverage that observation about loop branches to do better. Loop branches are, by definition, backwards (go back to previous code). So we can design a branch predictor which predicts "taken" for backwards and "not taken" for forwards. The compiler can then use this information to encode what it thinks about forwards branches (that is, making the not-taken branch the one it thinks is more likely). Let's say that this might get us to 80% accuracy.

$$(0.8 + 0.8 \times 0.2) \times 1 \text{ cycle} + (0.2 \times 0.2) \times 20 \text{ cycles} = 1.76 \text{ cycles}.$$

The PPC 601 (1993) and 603 used this scheme.

Going dynamic: using history for branch prediction. So far, we will always make the same prediction at each branch—known as a *static* scheme. But we can do better by using what recently happened to improve our predictions. This is particularly important when program execution contains distinct phases, with distinct behaviours. We therefore move to *dynamic* schemes.

Once again, let's start with the simplest possible thing. For every branch, we record whether it was taken or not last time it executed (a 1-bit scheme). Of course, we can't store all branches. So let's use the low 6 bits of the address to identify branches. Doing so raises the prospect of *aliasing*: different branches (with different behaviour) map to the same spot in the table.

We might get 85% accuracy with such a scheme.

$$(0.8 + 0.85 \times 0.2) \times 1 \text{ cycle} + (0.15 \times 0.2) \times 20 \text{ cycles} = 1.57 \text{ cycles}.$$

At the cost of more hardware, we get noticeable performance improvements. The DEC EV4 (1992) and MIPS R8000 (1994) used this one-bit scheme.

Two-bit schemes. What if a branch is almost always taken but occasionally not taken (e.g. TTTTTTNTTTT)? We get penalized twice for that misprediction: once when we mispredict the not taken, and once when we mispredict the next taken. So, let's store whether a branch is "usually" taken, using a so-called 2-bit saturating counter.

Every time we see a taken branch, we increment the counter for that branch; every time we see a not-taken branch, we decrement. Saturating means that we don't overflow or underflow. We instead stay at 11 or 00, respectively.

If the counter is 00 or 01, we predict "not taken"; if it is 10 or 11, we predict "taken".

With a two-bit counter, we can have fewer entries at the same size, but they'll do better. It would be reasonable to expect 90% accuracy.

$$(0.8 + 0.9 \times 0.2) \times 1 \text{ cycle} + (0.1 \times 0.2) \times 20 \text{ cycles} = 1.38 \text{ cycles}.$$

This was used in a number of chips, from the LLNL S-1 (1977) through the Intel Pentium (1993).

Two-level adaptive, global. We're still not taking patterns into account. Consider the following `for` loop.

```
for (int i = 0; i < 3; ++i) {
  // code
}
```

The last three executions of the branch determine the next direction:

```
TTT => N
TTN => T
TNT => T
NTT => T
```

Let's store what happened the last few times we were at a particular address—the *branch history*. From a branch address and history, we derive an index, which points to a table of 2-bit saturating counters. What's changed from the two-bit scheme is that the history helps determine the index and hence the prediction.

After we take a branch, we add its direction to the history, so that the next lookup maps to a different table entry.

This scheme might give something like 93% accuracy.

$$(0.8 + 0.93 \times 0.2) \times 1 \text{ cycle} + (0.07 \times 0.2) \times 20 \text{ cycles} = 1.27 \text{ cycles}.$$

The Pentium MMX (1996) used a 4-bit global branch history.

Two-level adaptive, local. The change here is that the CPU keeps a separate history for each branch. So the branch address determines which branch history gets used. We concatenate the address and history to get the index, which then points to a 2-bit counter again. We are starting to encounter diminishing returns, but we might get 94% accuracy:

$$(0.8 + 0.94 \times 0.2) \times 1 \text{ cycle} + (0.06 \times 0.2) \times 20 \text{ cycles} = 1.23 \text{ cycles}.$$

The Pentium Pro (1996), Pentium II (1997) and Pentium III (1999) use this.

gshare. Instead of concatenating the address and history, we can xor them. This allows us to use more bits for both the history and address. This keeps the accuracy the same, but simplifies the design.

Other predictors. We can build (and people have built) more sophisticated predictors. These predictors could, for instance, better handle aliasing, where different branches/histories map to the same index in the table. But we'll stop here.

Summary of branch prediction. We can summarize as follows. Branch prediction enables pipelining and hence increased performance. We can create a model to estimate just how critical branch prediction is for modern processors. Fortunately, most branches are predictable now. Aliasing (multiple branches mapping to the same entry in a prediction table) can be a problem, but processors are pretty good at dealing with that too.

Side-channel attacks

A few years ago, a lot happened in terms of exploiting the hardware of CPU architectures to get access to privileged data, and unfortunately these things have performance implications!

Cache Attacks

In early 2018, the Spectre [KGG+18] and Meltdown [LSG+18] attacks were disclosed. These attacks leverage performance features of modern CPUs to break process isolation guarantees—in principle, a process shouldn't be able to read memory that belongs to the kernel or to other processes.

The concept of cache side-channel attacks has been known for a while. If an attacker can get some memory loaded into the cache, then it can extract that memory using a cache side-channel attack.

Spectre and Meltdown can cause privileged memory to be loaded into the cache, and then extracted using a cache side-channel attack. We'll talk about Spectre (variant 2), since it attacks branch prediction in particular. My explanation follows [Mas18] by Jon Masters of RedHat. However, you should now have enough background to follow the Google Project Zero description at [Hor18].

We know that at a branch, the CPU will start speculatively executing code at the inferred target of the branch. To exploit this vulnerability, the attack code convinces the CPU to speculatively execute code of its choice. The speculatively-executed attack code reads secret data (e.g. from the hypervisor kernel) and puts it in the cache. Once the data is in the cache, the attack proceeds by extracting the data from the cache.

Unfortunately, the only mitigation thus far involved additional security measures (in hardware and software) that unfortunately result in lower performance in program execution.

Hyperthreading attacks

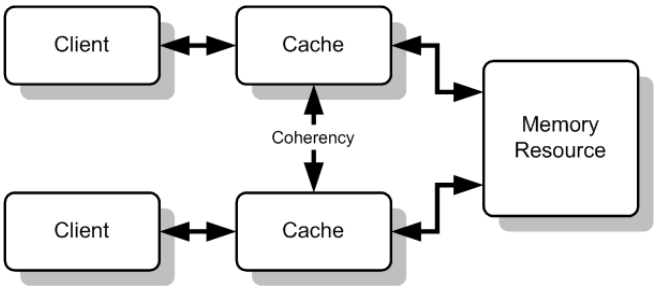
Multiprocessor (multicore) processors have some hardware that tries to keep the data consistent between different pipelines and caches (as we saw in the video). More processors, more threads means more work is necessary to keep these things in order. We will discuss cache coherence soon, but about hyperthreading... it turns out this is vulnerable too.

Remember that in hyperthreading, two threads are sharing the same execution core. That means they have hardware in common. Because of this, a thread can figure out what the other thread is doing by noticing its cache accesses and by timing how long it takes to complete operations. This is like sitting next to someone who's taking a multiple choice exam and noticing what answers they are choosing by how long it takes them to move their pencil down the page to fill in the correct circle. Yes, you have to be running on the same CPU as the victim, but still... Yikes!

Researchers discovered and published a paper [ABuH+18] detailing the attack and showing a practical implementation of it. In the practical example, a 384-bit secret key is (over time) completely stolen by another process. It seems likely that this will lead in the long term to slowdowns of existing hardware as Operating System patches will need to prevent threads from different processes from using the same core... And possibly the only long term solution is to not use hyperthreading at all... the performance implications of which are both obvious and significant.

8 — Cache Coherency

Cache Coherency



—Careless hx, CC BY-SA 4.0, via Wikimedia Commons

Today we'll look at what support the architecture provides for memory ordering, in particular in the form of cache coherence. We'll be talking about cache coherence strategies that work for CPUs, where we don't get much choice. But what we're going to talk about works equally well for something like `redis (redict)` in a situation where we have a distributed cache in software. In a software scenario we might get to choose the configuration that we want; when it comes to the CPU we get whatever the hardware designers have provided to us.

The problem is, of course, that each CPU likely has its own cache(s). If it does, then the data in these may be out of sync—the value that CPU 1 has for a particular piece of data might be different from the value that CPU 4 has. The simplest method, and a horrible solution, would be the ability to declare some read/write variables as being non-cacheable (is that a word? Uncacheable?...). The compiler and OS and such will require the data to be read from main memory, always. This will obviously result in lower cache hit ratios, increased bus traffic, and terrible, terrible performance. Let's avoid that. What we want instead is *coherency*.

Cache coherency means that (1) the values in all caches are consistent; and (2) to some extent, the system behaves as if all CPUs are using shared memory.

In modern CPUs with three or four levels of cache, we frequently find that the level 3 cache isn't much faster than going to main memory. But this level is where the cache coherency communication can take place. This can be by making the cache shared between the different CPUs. And the L4 cache is frequently used for sharing data with the integrated graphics hardware on CPUs that have this feature. But for the most part we will imagine that caches are not shared, and we have to figure out how to get coherency between them. This is the case with a L1/L2 cache in a typical modern CPU as they are unique to the given core (i.e., not shared).

Cache Coherence Example. We will use this example to illustrate different cache coherence algorithms and how they handle the same situation.

Initially in main memory: `x = 7`.

1. `CPU1` reads x, puts the value in its cache.
2. `CPU3` reads x, puts the value in its cache.

3. `CPU3` modifies `x := 42`
4. `CPU1` reads x . . . from its cache?
5. `CPU2` reads x. Which value does it get?

Unless we do something, `CPU1` is going to read invalid data.

Outside of a computing context, imagine you and several co-workers have some shared information, such as a meeting (in a specific room) in a shared online calendar (the one for the room). You (or anyone else) could make changes to this event. As mind-reading does not work, there are two ways that another invitee can know that something has changed: (1) they can check to see if anything has changed, or (2) they can be notified that a change has occurred.

The notification may contain the updated information in its entirety, such as "Event title changed to 'Discuss User Permissions and Roles'", or it may just tell you "something has changed; please check". In transportation, you can experience both. . . in the same day. I [JZ] was flying to Frankfurt and going to catch a train. Air Canada sent me an e-mail that said "Departure time revised to 22:00" (20 minute delay); when I landed the Deutsche Bahn (German railways) sent me an e-mail that said "Something on your trip has changed; please check and see what it is in the app". . . it was my train being cancelled. I don't know why they couldn't have e-mailed me that in the first place! It's not like I was any less annoyed by finding out after taking a second step of opening an app.

Regardless of which method is used, we have to pick one. Otherwise, we won't get the right answers.

Snoopy Caches. The simplest strategy is Snoopy caches [KMRS88]. No, not this kind of Snoopy (sadly):

It's called Snoopy because the caches are, in a way, spying on each other: they are observing what the other ones are doing. This way, they are kept up to date on what's happening and they know whether they need to do anything. They do not rely on being notified explicitly. This is a bit different from the transportation analogy, of course, but workable in a computer with a shared bus.

This is a distributed approach; no centralized state is maintained. Each cache with a copy of data from a block of main memory knows whether it is shared or not. All the CPUs are connected to a shared bus, and each CPU has its own cache controller. Whenever a CPU issues a memory write, the other CPUs are watching (colloquially, "snooping around") to observe if that memory location is in their cache. If so, the CPU will need to take action.

What does action mean? In the flight plus train example, both kinds of action occurred. The Air Canada action was *update*—the information about the flight departure time was changed from 21:40 to 22:00 and at the time of becoming aware of the change, I got the new value immediately. The Deutsche Bahn action was *invalidate*—the information about the train was changed, but I didn't know what had changed. All that I really knew is that the old information I had was out of date. When I needed that information again, I had to go get it myself from the source (their app). Either action (noting down the new, or knowing that what I have is out of date) is adequate for ensuring that I have the most up to date information. You may have a preference on which one you think is better, but unfortunately this is not your decision, neither as a user of the hardware of the computer nor as a person who wants to travel by plane or train.

Write-Through Caches

Let's put that into practice using write-through caches, the simplest type of cache coherence.

- All cache writes are done to main memory.
- All cache writes also appear on the bus.
- If another CPU snoops and sees it has the same location in its cache, it will either invalidate or update the data.

Invalidation is the most common protocol. It means the data in the cache of other CPUs is not updated, it's just noted as being out of date (invalid). Normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**). This kind of thing happens if you're just doing `x = 42;`—it doesn't matter what value of `x` was there before; you're just overwriting it.

If we want to do a read and there's a miss, we can ask around the other caches to see who has the most recent cached version. This is a bit like going into a room and yelling "Does anybody have block...?", in some sort of multicast version of the card game "Go Fish". Regardless, the most recent value appears in memory, always, so if nobody else has it in cache (or they don't feel like sharing) you can get it from there.

There are also write broadcast protocols, in which case all versions in all caches get updated when there is a write to a shared block. But it uses lots of bandwidth and is not necessarily a good idea. It does, however prevent the costly cache miss that follows an invalidate. Sadly, as we are mere users and not hardware architects, we don't get to decide which is better; we just have to live with whichever one is on the hardware we get to use. Bummer.

Write-Through Protocol. The protocol for implementing such caches looks like this. There are two possible states, **valid** and **invalid**, for each cached memory location. Events are either from a processor (**Pr**) or the **Bus**. Actions will be either a **Rd** (read) or **Wr** (write). We then implement the following state machine.

State	Observed	Generated	Next State
Valid	PrRd		Valid
Valid	PrWr	BusWr	Valid
Valid	BusWr		Invalid
Invalid	PrWr	BusWr	Valid
Invalid	PrRd	BusRd	Valid

Example. For simplicity (this isn't an architecture course), assume all cache reads/writes are atomic. [20] Using the same example as before:

Initially in main memory: `x = 7`.

1. `CPU1` reads x, puts the value in its cache. (valid)
2. `CPU3` reads x, puts the value in its cache. (valid)
3. `CPU3` modifies `x := 42`. (write to memory)
    - `CPU1` snoops and marks data as invalid.
4. `CPU1` reads x, from main memory. (valid)
5. `CPU2` reads x, from main memory. (valid)

Write-Back Caches

Let's try to improve performance. What if, in our example, `CPU3` writes to `x` 3 times in rapid succession? It's unpleasant to have to flush that to memory three times when we could do it only once. Let's try to delay the write to memory as long as possible. At minimum, we need support in hardware for a "dirty" bit, which indicates the our data has been changed but not yet been written to memory.

[20] If you're a hardware person, this line probably makes you cry. There's a whole lot that goes into making this work. There are potential write races, which have to be dealt with by contending for the bus and then completing the transaction, possibly restarting a command if necessary. If we have a split transaction bus it's really ugly, because we can have multiple interleaved misses. And down the rabbit hole we go.

Write-Back Implementation. The simplest type of write-back protocol (MSI) uses 3 states instead of 2:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Shared**—location is unmodified, up-to-date with main memory; may be present in other caches (also up-to-date).
- **Invalid**—same as before.

The initial state for a memory location, upon its first read, is "shared". The implementation will only write the data to memory if another processor requests it. During write-back, a processor may read the data from the bus.

MSI Protocol. Here, bus write-back (or flush) is **BusWB**. Exclusive read on the bus is **BusRdX**.

State	Observed	Generated	Next State
Modified	PrRd		Modified
Modified	PrWr		Modified
Modified	BusRd	BusWB	Shared
Modified	BusRdX	BusWB	Invalid
Shared	PrRd		Shared
Shared	BusRd		Shared
Shared	BusRdX		Invalid
Shared	PrWr	BusRdX	Modified
Invalid	PrRd	BusRd	Shared
Invalid	PrWr	BusRdX	Modified

MSI Example. Using the same example as before:

Initially in main memory: `x = 7`.

1. `CPU1` reads x from memory. (BusRd, shared)
2. `CPU3` reads x from memory. (BusRd, shared)
3. `CPU3` modifies `x = 42`: (PrWr, shared)
    - Generates a BusRdX.
    - `CPU1` snoops and invalidates x.
4. `CPU1` reads x: (PrRd, invalid)
    - Generates a BusRd.
    - `CPU3` writes back the data and sets x to shared.
    - `CPU1` reads the new value from the bus as shared.
5. `CPU2` reads x from memory. (BusRd, shared)

An Extension to MSI: MESI

The most common protocol for cache coherence is MESI. This protocol adds yet another state:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Exclusive**—only this cache has a valid copy; main memory is **up-to-date**.
- **Shared**—same as before.
- **Invalid**—same as before.

MESI allows a processor to modify data exclusive to it, without having to communicate with the bus. MESI is safe. The key is that if memory is in the E state, no other processor has the data. The transition from E to M does not have to be reported over the bus, which potentially saves some work and reduces bus usage.

MESIF: Even More States! MESIF (used in latest i7 processors): **Forward**—basically a shared state; but, current cache is the only one that will respond to a request to transfer the data.

Hence: a processor requesting data that is already shared or exclusive will only get one response transferring the data. Under a more simple MESI scheme you could get multiple caches trying to answer, with leads to bus arbitration or contention. The existence of a F state permits more efficient usage of the bus.

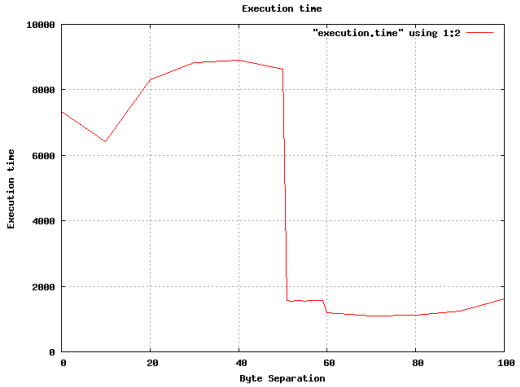
False Sharing

False sharing is something that happens when our program has two unrelated data elements that are mapped to the same cache line/location. That can be because of bad luck (hash collision kind of problem), but it often takes place because the data elements are stored consecutively. Let's consider an example from [Men08]:

```
char a[10];
char b[10];
```

These don't overlap, but are almost certainly allocated next to each other in memory. If a thread is writing to `a` and they share a cache line or block, then `b` will be invalidated; any CPU working on `b` will be forced to fetch the newest value from memory. This can be avoided by forcing some separation between these two arrays. One way would be to heap allocate both arrays. Usually if you do this you will find that they are not both located at the same location (but it's not guaranteed). So the other alternative is to make both arrays bigger than they need to be such that we're sure they don't overlap.

Consider the graph below that shows what happens in a sample program reading and writing these two arrays, as you increase the size of arrays `a` and `b` (noting that byte separation of 11 means they are adjacent; anything less than that and they overlap). This does waste space, but is it worth it?



Execution time graph showing 5x speedup by "wasting" some space [Men08].

At separation size 51 there is a huge drop in execution time because now we are certainly putting the two arrays in two locations that do not have the false sharing problem. Is wasting a little space worth it? Yes! Also—putting these arrays in a struct and padding the struct can also help with enabling future updates to the struct.

Software Implementation

A previous exam question on caching asked students to write a pseudocode description of behaviour for a distributed software cache that uses the MESI states and has write-back behaviour. This cache is for data items retrieved from a database, so if the item is not in any node's cache, write down `retrieve item i from database`.

You can assume the cache to be of significant size and while the question said you can assume that the least-recently-used (LRU) algorithm is used for replacement, you don't really have to consider replacement in this situation at all. As a practice problem for consideration, think about what modification(s) you would need to make for that scenario.

As the cache is distributed, we do need to consider what happens if a node comes online and joins the cluster, and what happens if a node is going to shut down and leave the cluster. You may ignore situations like crashes or network outages and you can assume all sent messages are reliably delivered/received.

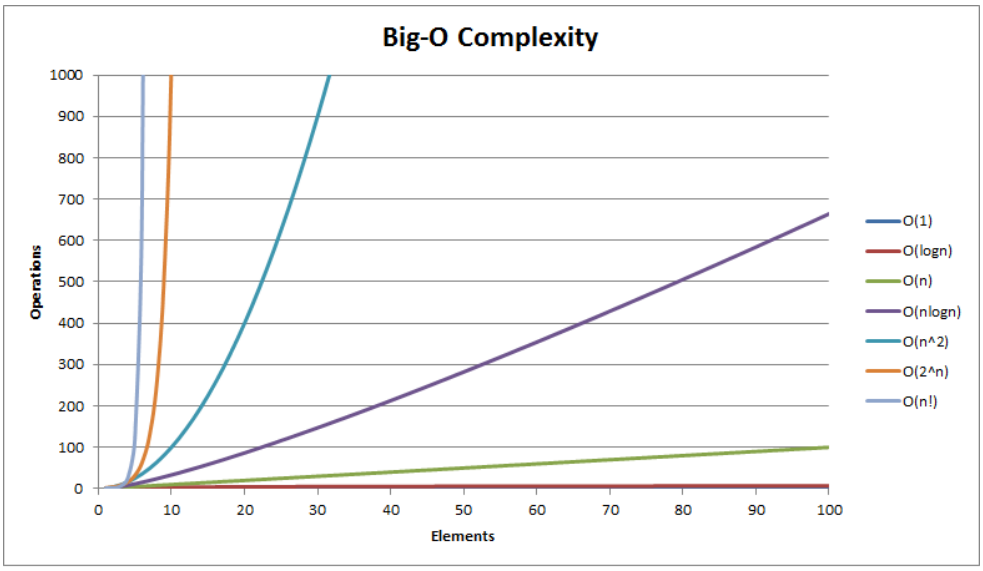
```
Current Node Shutdown {
  for i in items {
    if i is in state M {
      write i to the database
    }
  }
  for node n in known_nodes {
    send leaving message to n
  }
}

Node Leaves ( node n ) {
  Remove n from known_nodes
}

Get Item ( item i ) {
  if i is in local cache {
    return i
  } else {
    for node n in known_nodes {
      if n has item i
        add to cache ( i )
        set i state to S
        return i
      }
    }
  }
  retrieve i from database
  add to cache ( i )
  set i state to E
  return i
}
```

```
Other Node Searching ( item i ) {
  if i is in local cache {
    if i is in state M {
      write i to the database
      set i state to S
      return i
    } else if i is in state E {
      set i state to S
      return i
    } else if i is in state S {
      return i
        }
  }
  return null //Or other indicator of not-found
}

Update Item ( item i ) {
  if i is in local cache {
    if i is in state S {
      for node n in known_nodes {
        send invalidate i message to n
      }
      set i state to M
    } else if i is in state E {
      set i state to M
      return
    } else if i is in state M {
      return // Nothing to do
    }
  }
  add i to the cache in state M
}

Invalidate ( item i ) {
  if i is in local cache {
    set i state to I
  }
}
```

9 — Algorithms, Concurrency, and Parallelism

Algorithms

Remember from ECE 250/CS 138 that we often care about the worst case run-time performance of the algorithm. A bad algorithm has terrible runtime characteristics for a large data set. Trying to do an insertion sort on a small array is fine (actually... recommended); doing it on a huge array is madness. Choosing a good algorithm is very important if we want it to scale. But you know that already; you're not in a 4th year engineering course to have me tell you that you can use quicksort rather than bubble sort.



Big-O Complexity comparison from [R+15]

Sorting is probably a bad example, though, because in a non-university-course situation, you almost certainly use a library- or language-provided implementation and you can just call `sort()` on the collection and it's done for you. But what about other situations?

While I [JZ] am not generally a fan of leetcode-style interviews as a way of determining aptitude for a job, there are some good ideas in how to approach that sort of interview that *can* be applied in real life. The objection I have is not that this is something you *never* do – just that it happens somewhat rarely and much more time will be spent on architecture or just figuring out what the needs of the user are. But anyway! Here's the basics of what I advise people when preparing for such an interview, and it applies well to non-interview situations.

The best place to start is likely a simple solution, which may be brute-force. In many scenarios, that's adequate because the code in question is not performance-critical or the order of magnitude of n in the Big-O notation is sufficiently small that it doesn't matter. Given a working (correct) implementation, if you then discover it's not working performance-wise, the next step is to refine it.

Refinement here means to improve the algorithm you've got. The book *Cracking the Coding Interview* [McD15] uses the acronym BUD for this: Bottlenecks, Unnecessary Work, Duplicated Work. Let's look at those things a bit.

A bottleneck is exactly what it sounds like, a rate-limiting part of the code. If we could improve that, we'd get the most benefit. If your code has multiple phases (do X, then do Y, then do Z), figuring out which of these is the slowest one helps. If X is load data, Y is to do some transformation of it, and Z is send the data over the network – optimizing sending doesn't accomplish much if the slowest part is loading data.

Unnecessary work is also straightforward to define: any work that we did that isn't necessary. If we found the data element we need in the array we can stop looking now: anything else is wasted effort. That sort of thing doesn't change the algorithmic complexity, but it does reduce the total time of execution. Or another example would be considering data where we can tell in advance it's not going to work out – like a real-time system that won't schedule a task that has no hope of meeting its deadline.

And finally, duplicated work is anything more than once. Re-computing a value when you could remember that result instead is an easy example – even if the compiler might bail you out there. A more realistic example is if you repeatedly search an array, maybe putting all the data in a hashmap first saves a lot of effort.

In a coding-interview situation, you have a well-defined set of pieces of information. Chances are, you need to use all of them to find optimal solution. In other situations, it's not provided for you in the same way, so you may need to investigate. As an example, a coding problem might say that the array is sorted on the `id` field in ascending order. That might be true of the data you're getting in the function you're writing, but you may have to do some research or investigation to find out if that's the case, or maybe you need to be the change you want to see and add a call to sort the data.

Most of the discussion of Big-O complexity assumes a large enough n in the situation that the other terms don't matter. That's fine for an interview, but is not always true in real life. Sorting a large array or putting it into a Hashmap is likely to be optimal if you intend to use it many times, but maybe is not worthwhile if we just need to search it once.

It is also possible, as (sometimes) in a coding interview, to ask for help, if stuck. Another look or the eyes of a more senior colleague can help you spot an improvement or idea (or just help you get there faster).

Remember also that algorithmic complexity improvements always have limits. For ECE 459 at the end of the term, to successfully mark the final exam, the teaching team really does have to look at every question of every exam paper. No amount of cleverness in the process – switching to Crowdmark rather than marking on paper, for example – can get around the fact that the only way to properly mark the exam is to look at every single question as answered by each student. So this has linear runtime and we can never do any better than that.

Accidentally Quadratic

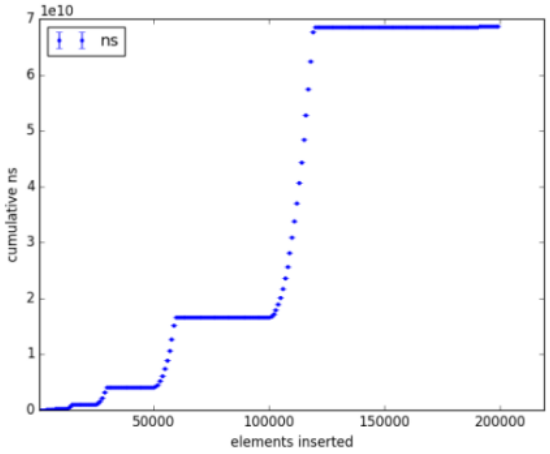
Given that many problems really are linear at the core (take an item, do something with it, go on to the next), a problem arises when we combine two linear things and get quadratic behaviour. Oh no! This is a situation that can be described as "accidentally quadratic".

There are a number of good examples at `https://accidentallyquadratic.tumblr.com/` if you are interested in seeing some more, though they are in different languages and the most recent post seems to be from mid-2019. To find a Rust-specific example, I had to go pretty far back in the archives and came up with this one from 2016(!): `https://accidentallyquadratic.tumblr.com/post/153545455987/rust-hash-iteration-reinsertion`. Let's recap their explanation:

Rust's hash tables used a strategy called *Robin-Hood Hashing*[21], which is based on open addressing with linear probing. Remember that open addressing is finding an alternate location in the event of a collision, rather than chaining and linear probing means you start from the bucket we should land in and move forward until you find a free space. The Robin-Hood part says if we have an item that's farther from its intended bucket than the current item, swap them. So if the order of items is 2 - 0 - 1, we'll swap until 0 - 1 - 2. This reduces the variance of items – how far away from where they "should" be on average.

[21] See this paper! `https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf`

Suppose you want to copy the data to a new hash table. To illustrate the problem, start by copying the data to a table half the size of the original. This works fine for the first half, but then as the second hash table is getting full we have to scan longer and longer to find the right place for it to go. Because this linear scan of finding a free bucket is already within the linear loop of "copy all items from table one to table two" we have indeed found an accidentally quadratic situation. When the second table is full enough to trigger a resize, then the problem goes away... at least until the table is almost full. See the time taken diagram:



That certainly looks quadratic. Not all the time, of course, but that doesn't matter. Algorithmic analysis is all about the worst-case scenario and sometimes when we insert things into this table it's quadratic and that's bad.

Not convincing? A simpler version of this looks something like: given a large number of elements to insert to an almost-full hash table, the runtime becomes quadratic because for each element you have to do a linear search to find where to insert it.

Now... please don't over-index on this one example to the point that you're arguing with your interviewer that *Well, actually* a hashmap doesn't help and that your solution without one is better. It's just that the real world has nuance and unexpected interactions that mean sometimes the normally-best approach is not the best this time.

If we can't do any better than linear (or maybe even quadratic), what do we do? Going back to the exam marking example: if I [JZ] had to mark 400+ exams myself, sequentially, I don't know how I'd do it... Or at least it would take so long that I'd get a nasty letter from the Registrar's Office about grades being due. I've got help, though, from the rest of the teaching team (co-instructors, TAs, LI). We divide up the work to finish on time. So – parallelism!

Concurrency and Parallelism

Concurrency and parallelism both give up the total ordering between instructions in a sequential program, for different purposes. We're going to focus on threads, but if you need a review of the details and differences of processes vs threads, you might like to read `https://www.purplealienplanet.com/node/50`.

Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, with no ordering between these components, like graphical user interfaces.

Parallelism. We're studying parallelism in this class, where we try to do multiple things at the same time in an attempt to increase throughput. Concurrent programs may be easier to parallelize.

Limits to parallelization

I mentioned briefly in Lecture 1 that programs often have a sequential part and a parallel part. We'll quantify this observation today and discuss its consequences.

Amdahl's Law. One classic model of parallel execution is Amdahl's Law. In 1967, Gene Amdahl argued that improvements in processor design for single processors would be more effective than designing multi-processor systems. Here's the argument. Let's say that you are trying to run a task which has a serial part, taking fraction S, and a parallelizable part, taking fraction $P = 1 - S$. Define T_s to be the total amount of time needed on a single-processor system. Now, moving to a parallel system with N processors, the parallel time T_p is instead:

$$T_p = T_s \cdot (S + \frac{P}{N}).$$

As N increases, T_p is dominated by S, limiting potential speedup.

We can restate this law in terms of speedup, which is the original time T_s divided by the sped-up time T_p:

$$\text{speedup} = \frac{T_s}{T_p} = \frac{1}{S + P/N}.$$

Replacing S with $(1 - P)$, we get:

$$\text{speedup} = \frac{1}{(1 - P) + P/N},$$

and

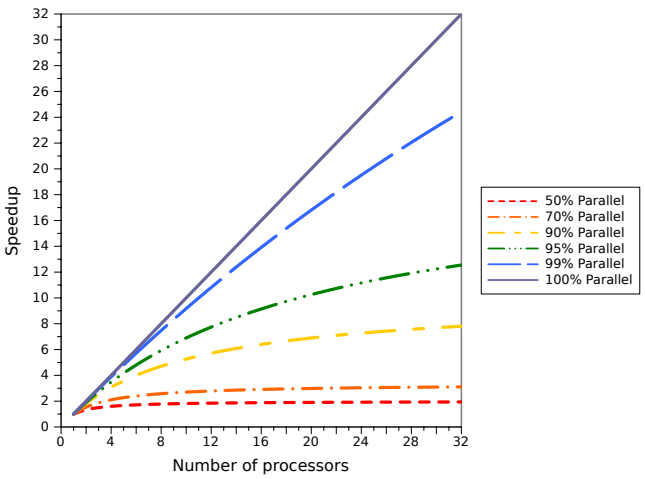
$$\text{max speedup} = \frac{1}{(1 - P)},$$

since $\frac{P}{N} \to 0$.

Plugging in numbers. If $P = 1$, then we can indeed get good scaling; running on an N-processor machine will give you a speedup of N. Unfortunately, usually $P < 1$. Let's see what happens.

P	speedup ($N = 18$)
1	18
0.99	~ 15
0.95	~ 10
0.5	~ 2

Graphically, we have something like this:



Amdahl's Law tells you how many cores you can hope to leverage in an execution given a fixed problem size, if you can estimate P.

Let us consider an example from [HZMG15]: Suppose we have a task that can be executed in 5 s and this task contains a loop that can be parallelized. Let us also say initialization and recombination code in this routine requires 400 ms. So with one processor executing, it would take about 4.6 s to execute the loop. If we split it up and execute on two processors it will take about 2.3 s to execute the loop. Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s. Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%. Applying the formula, we get the following run times:

Processors	Run Time (s)
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

Empirically estimating parallel speedup P. Assuming that you know things that are actually really hard to know, here's a formula for estimating speedup. You don't have to commit it to memory:

$$P_{\text{estimated}} = \frac{\frac{1}{\textit{speedup}} - 1}{\frac{1}{N} - 1}.$$

It's just an estimation, but you can use it to guess the fraction of parallel code, given N and the speedup. You can then use $P_{\text{estimated}}$ to predict speedup for a different number of processors.

Consequences of Amdahl's Law. For over 30 years, most performance gains did indeed come from increasing single-processor performance. The main reason that we're here today is that, as we saw a few classes ago, single-processor performance gains have hit the wall.

By the way, note that we didn't talk about the cost of synchronization between threads here. That can drag the performance down even more.

Amdahl's Assumptions. Despite Amdahl's pessimism, we still all have multicore computers today. Why is that? Amdahl's Law assumes that:

- problem size is fixed (read on);
- the program, or the underlying implementation, behaves the same on 1 processor as on N processors; and
- that we can accurately measure runtimes—i.e. that overheads don't matter.

Generalizing Amdahl's Law. We made a simplification, which was that programs only have one parallel part and one serial part. Of course, this is not true. The program may have many parts, each of which we can tune to a different degree.

Let's generalize Amdahl's Law:

- $f_1, f_2, \ldots, f_n$: fraction of time in part n
- $S_{f_1}, S_{f_n}, \ldots, S_{f_n}$: speedup for part n

Then,

$$\textit{speedup} = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + \ldots + \frac{f_n}{S_{f_n}}}.$$

Example. Consider a program with 4 parts in the following scenario:

		Speedup	
Part	Fraction of Runtime	Option 1	Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

(Note: these speedups don't have to be speedups from parallelization.)

We can implement either Option 1 or Option 2. Which option is better?

"Plug and chug" the numbers:

- **Option 1.**

$$\textit{speedup} = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

- **Option 2.**

$$\textit{speedup} = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

A more optimistic point of view

In 1988, John Gustafson pointed out[22] that Amdahl's Law only applies to fixed-size problems, but that the point of computers is to deal with bigger and bigger problems.

In particular, you might vary the input size, or the grid resolution, number of timesteps, etc. When running the software, then, you might need to hold the running time constant, not the problem size: you're willing to wait, say, 10 hours for your task to finish, but not 500 hours. So you can change the question to: how big a problem can you run in 10 hours?

According to Gustafson, scaling up the problem tends to increase the amount of work in the parallel part of the code, while leaving the serial part alone. As long as the algorithm is linear, it is possible to handle linearly larger problems with a linearly larger number of processors.

Of course, Gustafson's Law works when there is some "problem-size" knob you can crank up. As a practical example, observe Google, which deals with huge datasets.

Software Design Issues: Will it Parallelize?

Locking and Synchronization Points. Think back to a concurrency course and the discussion of locking. We'll be coming back to this subject before too long. But for now, suffice it to say, that the more locks and locking we need, the less scalable the code is going to be. You may think of the lock as a resource. The more threads or processes that are looking to acquire that lock, the more "resource contention" we have, and the more waiting and coordination are going to be necessary. We're going to revisit the subject of wise use of locks in more detail soon.

The previous paragraph applies as well to other concurrency constructs like semaphores, condition variables, etc. Any time a thread is forced to wait is going to be a limitation on the ability to parallelize the problem.

[22] http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html

Memory Allocators. Assuming we're not working with an embedded system where all memory is statically allocated in advance, there will be dynamic memory allocation. The memory allocator is often centralized and may support only one thread allocating or deallocating at a time. This means it does not necessarily scale very well. There are, however, some techniques for dynamic memory allocation that allow these things to work in parallel.

Overhead. A first implementation might involve starting a thread for a task, then destroying it when it is complete. If there are many tasks and tasks are short-lived, then the fraction of time creating and destroying the threads may be significant.

But that's not the only way. We can have a pool of workers. The workers are created once and only once. Then the application just submits units of work, and then on the other side these units of work are allocated to workers. The number of workers will scale based on the available hardware. This is neat as a programming practice: as the application developer we don't care quite so much about the underlying hardware. Let the operating system decide how many workers there should be, to figure out the optimal way to process the units of work.

Suppose you have to decide, though, how many threads should you create. This depends on which resources your threads use; if you are writing computationally-intensive threads, then you probably want to have fewer threads than the number of virtual CPUs. You can also use Amdahl's Law to estimate the maximum useful number of threads, as discussed previously.

Here's a longer discussion of thread pools:

`http://www.ibm.com/developerworks/library/j-jtp0730.html`

Modern languages provide thread pools; Java's `java.util.concurrent.ThreadPoolExecutor` [Ora10], C#'s `System.Threading.ThreadPool` [Cor05], and GLib's `GThreadPool` [The15] all implement thread pools. There's a Rust crate called `threadpool`. You can obviously write your own.

Here's a quick Rust program in which we use the threadpool crate to take away some of the complexity.

```
/*
[dependencies]
threadpool = "1.0"
thread-id = "4.0.0"
*/

use std::collections::VecDeque;
use std::sync::{Arc, Mutex};
use threadpool::ThreadPool;

fn main() {
    let pool = ThreadPool::new(8);
    let queue = Arc::new(Mutex::new(VecDeque::new()));
    println!("main␣thread␣has␣id␣{}", thread_id::get());

    for j in 0 .. 4000 {
        queue.lock().unwrap().push_back(j);
    }
    queue.lock().unwrap().push_back(-1);

    for _ in 0 .. 4 {
        let queue_in_thread = queue.clone();
        pool.execute(move || {
            loop {
                let mut q = queue_in_thread.lock().unwrap();
                if !q.is_empty() {
                    let val = q.pop_front().unwrap();
                    if val == -1 {
                        q.push_back(-1);
                        println!("Thread␣{}␣got␣the␣signal␣to␣exit.", thread_id::get());
                        return;
                    }
                    println!("Thread␣{}␣got:␣{}!", thread_id::get(), val);
                }
            }
```

```
        });
    }
    pool.join();
}
```

It's important to note that when we call the `execute` function, that is a job to be run, so if our thread pool has four workers we want to push the consume "job" on it four times. They will then run and each will try to consume numbers until they get to the -1 answer which is the termination signal.

If we wrote our own implementation where we spawned the threads using the spawn mechanism, joining each thread individually might be a bit of a pain.

This produces output that looks like:

```
main thread has id 4455538112
Thread 123145474433024 got: 0!
Thread 123145474433024 got: 1!
Thread 123145474433024 got: 2!

...

Thread 123145478651904 got: 3997!
Thread 123145478651904 got: 3998!
Thread 123145478651904 got: 3999!
Thread 123145476542464 got the signal to exit.
Thread 123145484980224 got the signal to exit.
Thread 123145474433024 got the signal to exit.
Thread 123145478651904 got the signal to exit.
```

10 — Software Architecture

System Design

When you look at the situation, you might find the reason that the runtime of some code is what it is results from the design of the larger system in which the data lives. For example, if we have to do a lot of network calls to get the data that's needed, that will increase the time to do the operation compared to getting all the data locally or in only one network call. If we can't get all the data in one shot, we have to loop, and that loop might easily turn what is otherwise linear runtime into another of the accidentally-quadratic examples.

System design interviews are another popular screening method for candidates in industry. I [JZ] like this better than the leetcode interviews in terms of understanding a candidate's ability to do the (typical) work of software development. Unlike most algorithm implementations which are solved problems (that is, there exist one or more optimal answers), a lot of system design problems are quite open-ended and a problem that isn't "read my mind" likely has multiple valid options that you can choose... if you can justify your choice appropriately. I'll be happy to talk to you outside of class about my advice about how to approach this sort of interview.

Whether you can change the data layout or the system architecture is very situation-dependent. It is generally unlikely that you will be able to convince your company to split up their five-year old monolith codebase because it would be faster in some scenarios. It's not necessarily that your argument is invalid, it's just that the opportunity cost is huge and such a cost has to be outweighed by sufficient customer value. Similarly, it might be optimal for your use case to change how the data is represented in the database, but that version might be worse for another, more common, use case and so the right decision is leave it alone.

Sometimes you will get input, or get to make the choice. Let's talk about that a little bit!

Choosing the Right Architecture

Software architecture courses cover this sort of topic in much more depth than this course has time for, but there is some time to think about the implications of architecture on the performance of the code. Most architecture decisions at the highest level – like the level of monolith vs microservices – are rarely made with performance in mind. We think about how best to represent the data or workflow... or what makes it easier for the developers to get work done. Alternatively, it's done with a wild guess about the performance situation: starting with microservices might be premised on the idea of scaling individual parts as needed sounds good but... will you really need it?

Since monoliths and microservices architecture have just been mentioned and will continue to feature in the rest of this topic, a quick explanation about these is warranted. A *monolith* architecture is one in which there is one software project that contains the functionality. There can (and should) be module structure and some amount of organization in the system, but it's one deployable unit. Communication between different parts of the monolith are just in-memory or via some internal API. A *microservices* architecture is one where there are many deployable units that work together via network communication. A medium-size or larger company almost certainly has some mixture of these things rather than a totally pure version of one of them.

Which of these architectures is the best depends on the needs of the project. Monolith codebases were the standard for a long time, then microservices came into vogue, then there was a backlash, and the cycle goes on. My general advice on this would be to start with a monolith architecture and only start breaking things off to microservices if that proves to be valuable later.

Down a Level. At the middle level, the decision of monolith or microservices is a given, but things like the design patterns come into play. Are we treating this as a producer-consumer problem? Blackboard-architecture? Message passing vs locking shared memory? Every approach has its pros and cons.

The discussing at this level is often around modelling the work to be done according to some convenient real-life analogy. Thinking of the processing of the data as an assembly line might be reasonable and a great way to make sure that the end result is correct. However, an implementation where the data moves from queue to queue could increase the overhead costs where the most efficient option is for a single thread to take the same work item through all steps.

Implementation. If we zoom in a bit more then we start to get into the implementation strategies for various parts of the system: do we have one thread? Spawn threads dynamically or use a thread pool?

Maybe the framework you chose has made some of those decisions for you and you just get to choose the configuration parameters. Such as if your framework for responding to REST API calls uses a threadpool, but you get to choose the number of threads in the pool. There's some analysis to do there and you can start by thinking about how many CPUs are available and the nature of the tasks (i.e., do they get blocked frequently on I/O). With that said, more threads isn't always better due to overhead costs and communication costs. Ultimately, you will need to experiment and see what happens. In a later topic, we will discuss more about APIs and rate limits.

Complexity is the Enemy. The complexity that we create is often the result of looking to other companies – particularly big companies that are leaders in the space – and thinking that their choices are the reasons they are successful and that this is the new best practice and a "silver bullet"[23] [Car24]. Looking at the industry leaders is instructive in many ways, but it's important to understand that many of their processes and choices are relevant for their size or scale and not for the rest of us.

Some complexity is unavoidable as a result of making software used by real people to do their work. Tax software contains a huge number of rules and their exceptions because the tax code (based on law, regulations, etc) has so much inherent complexity. You can write to your government and ask them to simplify it, but unless they actually go through with it, you have to implement all the rules. The other kind of complexity, however, is the ones that software developers bring themselves with things like the factory pattern and `VisitationArrangingOfficer` classes. If you would like to lose some sanity, please enjoy `FizzBuzzEnterpriseEdition`: `https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition`

That's too much complexity on purpose, as some sort of parody of the excessive complexity that "enterprise" software has. Minimizing the non-essential complexity is important and it might sound like the advice is to not have abstraction, but we often talk about how abstraction reduces the complexity. Which is it?

Levels of abstraction are often perceived as a good thing in software architecture. They allow us to identify commonalities and de-duplicate code. In fact, there's a saying by David J. Wheeler that goes "We can solve any problem by introducing an extra level of indirection." Wikipedia even calls it the Fundamental Theorem of Software Engineering. Mathematicians would hate this – that's not a theorem, it's a pithy statement. Honestly, I [JZ] hate it too! Every layer of indirection can easily add additional overhead and result in duplication of work because the right information is not available. It also slows down the speed of development while you have to just write more stuff to move data around and to keep track of what should be on what level (and defend against degradation of the structure). Use as much abstraction as necessary and no more.

This is an ancient problem that has hopefully died out in years past, but companies sometimes had dedicated people working on architecture. They don't write code or actually implement anything. They are sometimes called "architecture astronauts" and they create, in the words of Joel Spolsky "absurd, all-encompassing, high-level pictures of the universe that are all good and fine, but don't actually mean anything at all." [Spo01]. Such a person usually does not do too much damage directly, but they encourage over-engineering and overly-complex solutions and either one can make your code slow!

How does over-architecting the code make the code slow? More services and more layers means more moving data around, and longer chains of communication are slower than shorter ones. How about some specifics?

[23] In folklore, silver is deadly to werewolves. A silver bullet is a metaphor for a simple solution that instantly solves a very big problem.

Pitfalls

Let us consider a few specific things that we could do wrong (and what to do about them). Just like the clickbait articles: four huge architecture mistakes to avoid! Number two will shock you!

Excessive Network Calls. As previously referenced, reducing the number of network calls to get the data will be an improvement in the execution time. Network is slow and comes with unpredictable delays for various reasons. Each call has overhead to establish the connection, authenticate or validate the token[24], and unpack the request before any of the actual handling of it can take place. Thus, a lot of small requests is likely slower than one larger request. That may be partially offset by parallelization (like in asynchronous I/O situations), but not always.

The converse of that, though is that how much data is sent is also a factor in the length of time the communication takes. In a silly example, the app requests the whole table from the database server to the application and then searches and filters in memory. If the table is huge, sending all of it would take a long time. And that assumes that there's enough memory in the application and that the network call would not time out for being quite so large...

Worries about too many network calls can be used as an argument against a microservices architecture and favouring monoliths. It's a drawback of this architecture, to be sure, but it will never be the only criteria for deciding. Considering honestly the pros and cons of the architecture is normal, and if anyone ever tells you that their preferred solution has no drawbacks whatsoever, you should be very skeptical of that.

A common variant of this negative pattern is called the N+1 query problem. In short, it happens when you want to fetch a list of something, and then for each of those, fetch a list of related entities. Imagine you want to send e-mails to all the customers who have not yet paid their invoices. How to do that? Query for the customer ID of each invoices where the status is unpaid. Then, for each of those customer IDs, look up their e-mail address, and send the payment reminder e-mail to them.

At first glance that doesn't sound bad, because this is how you might carry out such an operation if doing it with printed paper invoices and contact info in your phone. But if there are 500 unpaid invoices, this approach results in 501 database queries. One to get the list of customer IDs, and then one each for every invoice to get the customer's e-mail. That's too many!

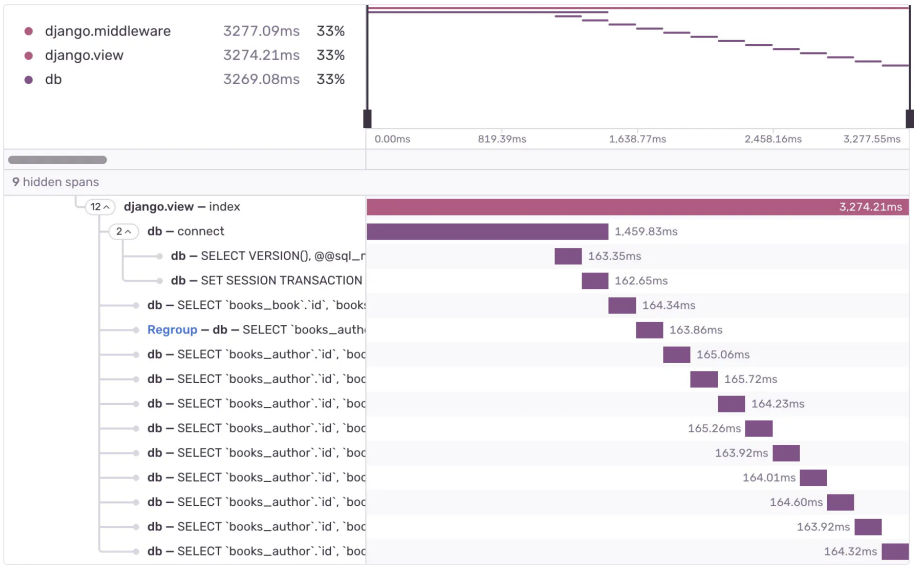
You might think that such a thing is easy to avoid – and it can be, if writing the SQL directly. We could use a join query to get the list of e-mails directly in one query, or use a `WHERE customer_id IN (...)` type of clause to get it down to two queries. I prefer the one query approach, if possible, but two is certainly better than 500. What did I mean about not writing the SQL directly?

If you're using some sort of ORM (Object-Relational Mapping), then you have some intermediate framework (e.g., Hibernate, Rails ActiveRecord) that turns some (Rust or other language) code into some database query statements that you didn't write yourself. The ORMs may unintentionally generate for you the N+1 query-problem variant of your query. It's not on purpose, but it is worth checking the generated SQL these produce just to be sure. Some ORMs let you write your own SQL; for others you want to pursue strategies like eager loading (that is, fetching related entities alongside the ones you asked for originally, even if you might not need the related ones) [Oza18].

Here's an example of what an N+1 query looks like when you look at it with a profiler, from the Sentry Docs:[25]

[24] You *are* including auth in your APIs, right? RIGHT?

[25] `https://docs.sentry.io/product/issues/issue-details/performance-issues/n-one-queries/`



The documentation for the Rust crate `juniper_eager_loading` provides a great example of how an N+1 query can result from a GraphQL schema. It builds on the Juniper package. We will not go too far into the details, but Juniper helps Rust backends use GraphQL.

In case you are not familiar with GraphQL, it is a data query language that lets the caller ask for the information that they want based on a known data schema. In a REST API, the verbs, endpoints, and the response shape are all determined by the implementer of the server responding. Perhaps the caller only wants to know the sum of unpaid invoice amount the account has, but if the API allows you only to get a list of unpaid invoices, you get back seven full invoices worth of data when all you cared about was the total field on that invoice. If you want to just ask about the total, well, add another endpoint... GraphQL is meant to avoid this by allowing you to ask just for what you need.

Okay, on to the example from the docs[26] which, to give proper credit, are written by David Pedersen (`https://github.com/davidpdrsn/`) which I have modified a bit. Here's an example GraphQL schema:

```
schema {
    query: Query
}

type Query {
    allUsers: [User!]!
}

type User {
    id: Int!
    country: Country!
}

type Country {
    id: Int!
    name: String!
}
```

And a caller executes the query:

```
query SomeQuery {
    allUsers {
        country {
            name
        }
    }
}
```

The naive implementation will run the following queries:

[26] `https://docs.rs/juniper-eager-loading/latest/juniper_eager_loading/index.html`

```
select * from users
select * from countries where id = ?
select * from countries where id = ?
select * from countries where id = ?
select * from countries where id = ?
...
```

But we actually would prefer...

```
select * from users
select * from countries where id in (?, ?, ?, ?)
```

That's what we would do in eager loading, but that's harder in GraphQL since it's not possible to predict in advance which queries that callers will actually choose. Alright, how does this library work? The core idea is changing how the data structures work to load information if needed but avoid it if it's not. Let's look at the default way you might create a struct for `User` based on the schema above:

```
struct User {
    id: i32,
    country_id: i32
}
```

That's good, but the only way to get the name for a given country associated with users is to load the country itself from the database because the user only has the country's ID and not the name. Let's look at the source's alternative approach that separates the GraphQL model from the actual Rust structures.

```
mod models {
    pub struct User {
        id: i32,
        country_id: i32
    }

    pub struct Country {
        id: i32,
        name: String
    }
}

struct User {
    user: models::User,
    country: HasOne<Country>,
}

struct Country {
    country: models::Country
}

enum HasOne<T> {
    Loaded(T),
    NotLoaded,
}
```

With that, we can respond to the query in a more efficient way. First, load the users, get a list of country IDs, then query the countries in that set of IDs, and match them up. That means replacing the `User.country` value of `HasOne::NotLoade` with `HasOne::Loaded(matching_country)`. Then we can just return the set of names after the second query. A little bit of data modelling goes a long way here. This solution may not be the most optimal or the fewest lines of code, but it does demonstrate a big improvement over the naive version.

Finally, another strategy that helps reduce the number of network calls is caching. We've discussed it at great length earlier in the course, so no need to repeat any of that.

Bottlenecks (Chokepoints). If your architecture results in repeatedly accessing the same data or same system, that can become a bottleneck on the way to completing work. One possible example of this that I [JZ] have seen is having an authorization service check the credentials on every API call internally in a microservices architecture. While that might be a good security policy to check every time, the communication costs in this scenario are high and it could end up overloading the authorization service (CPU) doing all the cryptographic calculations to check the credentials.

The solve for that particular problem is making it possible for a service to validate the credential without asking another system. A specific example of that is the `jwt` – JSON web token – which has some data (e.g., credentials) and is cryptographically signed with public-private key encryption so the server receiving the API call can check the credentials itself. Think of it as being like a passport: the government issues you the document and it has some security measures to prevent tampering; if you want to use your legitimate document to travel to another country the border control agents can validate the document without needing to do data exchange with the country that issued it. There are more levels at which the passport analogy for `jwt` works, but that's a digression.

The generalization of the solution above is to distribute the work to avoid the chokepoint: can other components take a part of it? Is the bottleneck part over-taking responsibility or would distributing it be putting work where it doesn't belong? That's worth considering.

Over-Taking Responsibility. The earlier example about searching and filtering in the application instead of the database also exemplifies another problem: over-taking responsibility. Even if the size of the table is so small that transmitting all of it over the network takes negligible time, why search and filter it in memory of the application? The database server has more information, like how the data is stored on disk or what index it can use to do the search. Therefore the most efficient approach is to let the DB do what it does best.

Another possibility is asking the backend system to do lots of work that would be better done in the frontend. Rendering the page in the backend may seem like a strange choice now, but it was the norm for a period of time. Moving that to the frontend or mobile app takes pressure off the backend. This does not work for everything: remember that the backend should not trust any input from the frontend, so validation still needs to happen in the backend to be sure.

Over-taking responsibility is sometimes a result of organizational or technical constraints, such as gatekeeping or the remote server being run by another company. So the fastest, or only, way to accomplish what you want is to do it yourself, even if that's inefficient in execution. It's also possible to get in this situation because procurement (buying things or signing up for some external service) is too difficult and too slow: build vs buy decisions might always favour build if it means you can have something this month instead of next year. Over-taking responsibility also tends to result in more painful change management and high levels of tech debt if people are afraid of touching a critical and fragile system.

There are no software-architecture decisions that can solve organizational problems like this, but you can hopefully negotiate for what you want, or perhaps build some intermediary system that helps divide the workload rather than just suffering.

Too Much Waiting. We've already covered asynchronous I/O, so at this point we are familiar with the idea that in some circumstances we can start an operation and not wait for the results before starting the next. We know that eliminates unnecessary waits and there's even a potential benefit of multiple I/O requests executing in parallel to bring down completion time. Still, there's more to explore on this idea.

Moving to an asynchronous model is also valuable in reducing *perceived* waits even if the actual execution time is no faster. Remember from a previous operating systems course the concept of *response time*. This is the time that it takes to get some result back. Emphasis on the word some in that sentence. The user has a better experience if we start returning partial results sooner and then load the rest later. How many websites have you been to where the page loads but different panels have the spinning loader while other stuff comes in?

The last way in which we can end up with too much waiting is every thread or process waiting to acquire a lock around some shared data. Imagine there is one `Arc<Mutex<Vector>>` and every thread wants to add data to it. We can make that much more efficient if we have one thread that owns the data and the other threads send their data via a channel or other message-passing mechanism to the owning thread and the owning thread takes those and processes them in turn. Is there overhead in establishing the channel and sending/receiving the data? Sure, but it is offset by reducing (eliminating) lock contention around the vector. Actually, let's talk some more in the next topic about the proper use of locks and the concept of lock contention.

11 — Use of Locks, Reentrancy

Appropriate Use of Locking

In previous courses you learned about locking and how it all works, then we did a quick recap of what you need to know about it. And perhaps you were given some guidance in the use of locks, but probably in earlier scenarios it was sufficient to just avoid all the bad stuff (data races, deadlock, starvation). That's important, but is no longer enough. Now we need to use locking and other synchronization techniques in a way that reduces their impact on performance.

I like to say that critical sections should be as large as they need to be but no larger. That is to say, if we have some shared data that needs to be protected by some mutual exclusion constructs, we need to consider carefully where to place the statements. They should be placed such that the critical section contains all of the shared accesses, both reads *and* writes, but also does not contain any extraneous statements. The ones that don't need to be there are those that don't operate on shared data.

If you are rewriting code from sequential to parallel, this can mean that a block of code or contents of a function need to be re-arranged to move some statements up or down so they are no longer in the critical section. Sometimes control flow or other very short statements might get swept into the critical section being created to make sure all goes as planned, so the rule is not absolute. However, such statements should be there rarely and only if the alternatives are worse.

Let's consider a short code example from the producer-consumer problem. In the course repository's code directory, the full code is available the original and modified forms. It makes sense to look over the original before we discuss how to improve it. We'll look at the consumer code:

```
for _j in 0 .. NUM_THREADS {
    // create consumers
    let spaces = spaces.clone();
    let items = items.clone();
    let buffer = buffer.clone();
    threads.push(
        thread::spawn(move || {
            for _k in 0 .. ITEMS_PER_THREAD {
                let permit = block_on(items.acquire());
                let mut buf = buffer.lock().unwrap();
                let current_consume_space = buf.consumer_count;
                let next_consume_space = (current_consume_space + 1) % buf.buffer.len();
                let to_consume = *buf.buffer.get(current_consume_space).unwrap();
                buf.consumer_count = next_consume_space;
                spaces.add_permits(1);
                permit.forget();
                consume_item(to_consume);
            }
        })
    );
}
```

When we used locks in C (or similar), it was easier to identify what's in the critical section, because we had explicit lock and unlock statements. The explicit unlock statement, especially, made it much clearer where it ends. Now, we don't consider the critical section over until the `MutexGuard` (returned by `lock()`) goes out of scope. And that happens here at the end of the iteration of the loop.

What I always say is to analyze this closure one statement at a time and look into which of these access shared variables. We're not worried about statements like locking or manipulating the semaphore, but let's look at the rest and decide if they really belong. Can any statements be removed from the critical section?

In a practical sense, the critical section needs to enclose anything that references `buf` and that's most of the statements, save those three at the end: adding permits to spaces, forgetting our current permit, and consuming the item. Rust is good about not letting you access shared data in an uncontrolled way, so we can feel more certain that there's nothing left out of the critical section that should be in there.

How do we end the critical section? We need to make our acquisition of the mutex go out of scope. The easiest way to do that is to use manual scoping:

```
    for _j in 0 .. NUM_THREADS {
        // create consumers
        let spaces = spaces.clone();
        let items = items.clone();
        let buffer = buffer.clone();
        threads.push(
            thread::spawn(move || {
                for _k in 0 .. ITEMS_PER_THREAD {
                    let permit = block_on(items.acquire());
                    let to_consume = {
                        let mut buf = buffer.lock().unwrap();
                        let current_consume_space = buf.consumer_count;
                        let next_consume_space = (current_consume_space + 1) % buf.buffer.len();
                        let to_consume = *buf.buffer.get(current_consume_space).unwrap();
                        buf.consumer_count = next_consume_space;
                        to_consume
                    };
                    spaces.add_permits(1);
                    permit.forget();
                    consume_item(to_consume);
                }
            })
        );
    }
```

You'll notice that we return the value `to_consume` out of the block, because it's needed outside the block and would otherwise not be in scope when passed to the function that consumes it. The whole purpose of this is to get the value outside of the block. Because it's a simple type, we'll copy it, but a more complex type would just have ownership transferred, so there isn't a large performance penalty here.

The other approach to making the `MutexGuard` go out of scope is to actually call `drop()` on it, which is effective in telling the compiler that it is time for this value to die. Calling `drop()` moves ownership of the MutexGuard to the `drop` function where it will go out of scope and be removed. Convenient! But manual scoping is nice too.

Let's see if it works! I applied a similar change the producer code as we just discussed about the consumer. And for the purposes of the test, I added some thread sleeps to the original and modified program so it appears that consuming or producing an item actually takes meaningful work. As usual, benchmarks are created with `hyperfine --warmup 1 -m 5 "cargo run --release"`. The un-optimized program takes about 2.8 seconds to run and the optimized program takes about 1.1 seconds. Certainly worth doing.

Remember, though, that keeping the critical section as small as possible is important because it speeds up performance (reduces the serial portion of your program). But that's not the only reason. The lock is a resource, and contention for that resource is itself expensive.

Locking Granularity

The producer-consumer example was a very specific instance of *lock granularity*: how much data is locked by a given lock. We have choices about the granularity of locking, and it is a trade-off (like always).

Coarse-grained locking is easier to write and harder to mess up, but it can significantly reduce opportunities for parallelism. *Fine-grained locking* requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc). Locks' extents constitute their *granularity*. In coarse-grained locking, you lock large sections

of your program with a big lock; in fine-grained locking, you divide the locks and protect smaller sections with multiple smaller locks.

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e., remaining race conditions). We'll assume there are adequate locks and that data accesses are protected.

Lock Overhead. Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention. Most locking time is wasted waiting for the lock to become available. We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Deadlocks. Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X'. ($X = X'$ is allowed).

Okay, in a formal sense, the four conditions for deadlock are:

1. **Mutual Exclusion**: A resource belongs to, at most, one process at a time.
2. **Hold-and-Wait**: A process that is currently holding some resources may request additional resources and may be forced to wait for them.
3. **No Preemption**: A resource cannot be "taken" from the process that holds it; only the process currently holding that resource may release it.
4. **Circular-Wait**: A cycle in the resource allocation graph.

Consider, for instance, two processors trying to get two locks.

Thread 1	**Thread 2**
`Get Lock 1`	`Get Lock 2`
`Get Lock 2`	`Get Lock 1`
`Release Lock 2`	`Release Lock 1`
`Release Lock 1`	`Release Lock 2`

Processor 1 gets Lock 1, then Processor 2 gets Lock 2. Oops! They both wait for each other. (Deadlock!).

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**. You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

```
let mut thing1 = l1.lock().unwrap()
let mut thing2 = l2.lock().unwrap()
// protected code
// locks dropped when going out of scope
```

```
let mut thing1 = l1.lock().unwrap()
let mut thing2 = l2.lock().unwrap()
// protected code
// locks dropped when going out of scope
```

This code will not deadlock: you can only get **l2** if you have **l1**. Of course, it's harder to ensure a consistent deadlock when lock identity is not statically visible. That is, if they don't always have the same names everywhere.

If we give a standard example from a textbook, we call the threads P and Q and they are attempting to acquire `a` and `b`. Thread Q requests `b` first and then `a`, while P does the reverse. The deadlock would not take place if both threads requested these two resources in the same order, whether `a` then `b` or `b` then `a`. Of course, when they have names like this, a natural ordering (alphabetical, or perhaps reverse alphabetical) is obvious.

We can certainly prove that consistent ordering does work and it's a proof by contradiction. If the set of all resources in the system is $R = \{R_0, R_1, R_2, ...R_m\}$, we assign to each resource R_k a unique integer value. Let us define this function as $f(R_i)$, that maps a resource to an integer value. This integer value is used to compare two resources: if a process has been assigned resource R_i, that process may request R_j only if $f(R_j) > f(R_i)$. Note that this is a strictly greater-than relationship; if the process needs more than one of R_i then the request for all of these must be made at once (in a single request). To get R_i when already in possession of a resource R_j where $f(R_j) > f(R_i)$, the process must release any resources R_k where $f(R_k) \geq f(R_i)$. If these two protocols are followed, then a circular-wait condition cannot hold [SGG13].

But I mentioned they might not have the same names everywhere. When locks travel with the data, this problem can arise. Consider the idea of a bank account and you want to transfer money from one to another. This will involve locking the sender account and locking the receiver account. And regardless of whether you say receiver first or sender first, there is the possibility of two concurrent transfers that mean we end up with a deadlock.

One thing that might work is making your structure somewhat different. Something like the account number is something that never changes, so you could leave that outside of the mutex and use that to determine an ordering, such as alphabetical ordering. That just means your struct is composed of the account number and the mutex surrounding another struct with the account data. Maybe a little weird, but it works.

Alternately, you can use trylock. Recall that Pthreads' `trylock` returns 0 if it gets the lock. But if it doesn't, your thread doesn't get blocked. Checking the return value is important, but at the very least, this code also won't deadlock: it will give up **l1** if it can't get **l2**.

```
loop {
    let mut m1 = l1.lock().unwrap();
    let m2 = l2.try_lock();
    if m2.is_ok() {
        *m1 += amount;
        *m2.unwrap() -= amount;
        break;
    } else {
        println!("try_lock_failed");
        // Go around the loop again and try again
    }
}
```

(Incidentaly, using trylocks can also help you measure lock contention.)

This prevents the hold and wait condition, which was one of the four conditions. A process attempts to lock a group of resources at once, and if it does not get everything it needs, it releases the locks it received and tries again. Thus a process does not wait while holding resources.

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (perhaps just one!). This is *coarse-grained locking*. It does have a couple of advantages:

- it is easier to implement;
- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

It also, however, has one big disadvantage in terms of programming for performance: your parallel program will quickly become sequential.

Example: Python (and other interpreters). Python puts a lock around the whole interpreter (known as the *global interpreter lock*)[27]. This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if one of the threads is waiting for I/O.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

You might think "this is stupid, who would ever do this?" Yet a lot of OS kernels do in fact have (or at least had) a "big kernel lock", including Linux and the Mach Microkernel. This lasted in Linux for quite a while, from the advent of SMP support up until sometime in 2011. As much as this ruins performance, correctness is more important. We don't have a class "programming for correctness" (software testing? Hah!) because correctness is kind of assumed. What we want to do here is speed up our program as much as we can while maintaining correctness...

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking*. The big advantage: it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Examples. Databases may lock fields / records / tables. (fine-grained $\rightarrow$ coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)

Reentrancy

Recall from a bit earlier the idea of a side effect of a function call. The trivial example of a non-reentrant C function:

```
int tmp;

void swap( int x, int y ) {
    tmp = y;
    y = x;
    x = tmp;
}
```

Why is this non-reentrant? Because there is a global variable `tmp` and it is changed on every invocation of the function. We can make the code reentrant by moving the declaration of `tmp` inside the function, which would mean that every invocation is independent of every other. And thus it would be thread safe, too.

Doing it wrong is highly discouraged by Rust as a language, because it doesn't want you to use global state (if it can help it) and it makes potential side effects pretty clear by requiring references to be annotated as mutable.

[27] `https://github.com/python/cpython/pull/116338`

Remember that in things like interrupt subroutines (ISRs) having the code be reentrant is very important. Interrupts can get interrupted by higher priority interrupts and when that happens the ISR may simply be restarted (or we're going to break off handling what we're doing and call the same ISR in the middle of the current one). Either way, if the code is not reentrant we will run into problems. Rust's ownership capabilities make it difficult for you to modify something that you should not modify with signal handlers, like calling some function that is not reentrant.

Side effects are sort of undesirable, but not necessarily bad. Printing to console is unavoidably making use of a side effect, but it's what we want. We don't want to call print reentrantly; interleaved print calls would result in jumbled output. Alternatively, restarting the print routine might result in some doubled characters on the screen.

The notion of purity is related to side effects. A function is pure if it has no side effects and if its outputs depend solely on its inputs. (The use of the word pure shouldn't imply any sort of moral judgement on the code). Pure functions should also be implemented to be thread-safe and reentrant.

Functional Programming and Parallelization

Interestingly, functional programming languages (by which I do NOT mean procedural programming languages like C), such as Haskell and Scala and so on, lend themselves very nicely to being parallelized. Why? Because a purely functional program has no side effects and they are very easy to parallelize. If a function is impure, its function signature will indicate so. Thus spake Joel[28]:

> *Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable.* [Spo05]

This assumes of course that there is no data dependency between functions. Obviously, if we need a computation result, then we have to wait. But the key is to write your code like mathematical functions: $f(x, y, z) \to (a, b, c)$.

Object oriented programming kind of gives us some bad habits in this regard: we tend to make a lot of `void` methods or those with no return type. In functional programming these don't really make sense, because if it's purely functional, then there are some inputs and some outputs. If a function returns nothing, what does it do? For the most part it can only have side effects which we would generally prefer to avoid if we can, if the goal is to parallelize things.

Rust nudges you towards functional-style programming. It discourages mutability of data, which encourages making functions arguments be either immutable references (hence not changed by the function they are passed to) or ownership transfer (meaning no concurrency). Internal mutability is a thing, but is somewhat discouraged.

[28] "Thus Spake Zarathustra" is a book by Nietzsche, and this was not a spelling error.

12 — Lock Convoys, Atomics, Lock-Freedom

Lock Convoys

We'd like to avoid, if at all possible, a situation called a *lock convoy*. This happens when we have at least two threads that are contending for a lock of some sort. And it's sort of like a lock traffic jam. A more full and complex description from [Loh05]:

> A lock convoy is a situation which occurs when two or more threads at the same priority frequently (several times per quantum) acquire a synchronization object, even if they only hold that object for a very short amount of time. It happens most often with critical sections, but can occur with mutexes, etc as well. For a while the threads may go along happily without contending over the object. But eventually some thread's quantum will expire while it holds the object, and then the problem begins. The expired thread (let's call it Thread A) stops running, and the next thread at that priority level begins. Soon that thread (let's call it Thread B) gets to a point where it needs to acquire the object. It blocks on the object. The kernel chooses the next thread in the priority-queue. If there are more threads at that priority which end up trying to acquire the object, they block on the object too. This continues until the kernel returns to Thread A which owns the object. That thread begins running again, and soon releases the object. Here are the two important points. First, once Thread A releases the object, the kernel chooses a thread that's blocked waiting for the object (probably Thread B), makes that thread the next owner of the object, and marks it as "runnable." Second, Thread A hasn't expired its quantum yet, so it continues running rather than switching to Thread B. Since the threads in this scenario acquire the synchronization object frequently, Thread A soon comes back to a point where it needs to acquire the object again. This time, however, Thread B owns it. So Thread A blocks on the object, and the kernel again chooses the next thread in the priority-queue to run. It eventually gets to Thread B, who does its work while owning the object, then releases the object. The next thread blocked on the object receives ownership, and this cycle continues endlessly until eventually the threads stop acquiring so often.

Why is it called a convoy? A convoy is when a grouping of vehicles, usually trucks or ships, travels all closely together. A freighter convoy, for example, might carry freight from one sea port to another. In this case, it means that the threads are all moving in a tight group. This is also sometimes called the "boxcar" problem: imagine that you have a train that is moving a bunch of boxcars along some railroad tracks. When the engine starts to pull, it moves the first car forward a tiny bit before it stops suddenly because of the car behind. Then the second car moves a bit, removing the slack between it and the next car. And so on and so on. The problem resembles this motion because each thread takes a small step forward before it stops and some other car then gets a turn during which it also moves forward a tiny bit before stopping. The same thing is happening to the threads and we spend all the CPU time on context switches rather than executing the actual code [Ost04].

This has a couple of side effects. Threads acquire the lock frequently and they are running for very short periods of time before blocking. But more than that, other, unrelated threads of the same priority get to run for an unusually large percentage of the (wall-clock) time. This can lead you to thinking that some other process is the real offender, taking up a large percentage of the CPU time. In reality, though, that's not the culprit. So it would not solve the problem if you terminate (or rewrite) what looks like offending process.

Unfair Locks. With that in mind, in Windows Vista and later versions, the problem is solved because locks are unfair. Unfair sounds bad but it is actually better to be unfair. Why? The Windows XP and earlier implementation of locks, which is fair, is a good explanation of why can go wrong. In XP, if A unlocks a lock ℓ, and there is a thread B waiting, then B gets the lock, it is no longer blocked, and when it wakes up, B already owns the lock. This is fair in the sense that there was no period during which the lock was available; therefore it could not be "stolen" by some other thread that happened to come along at the right (or perhaps wrong) time [Duf06]. (Specifically, if the OS chooses who gets the lock among all the waiting threads randomly, then that's fair.)

Fairness is good, right? But this means there is a period of time where the lock is held by B, but B is not running. In the best-case scenario, after A releases the lock, then there is a thread switch (the scheduler runs) and the context switch time is (in Windows, anyway, according to [Duf06]) on the order of 4 000-10 000 cycles. That is a fairly long time but probably somewhat unavoidable. If, however, the system is busy and B has to go to the back of the line it means that it might be a long time before B gets to run. That whole time, it is holding onto the lock. No one else can acquire ℓ. Worse yet, a thread C might start processing, request ℓ, and then we have to context switch again. That is a lock convoy.

Unfair locks help with lock convoys by not giving the lock to B when A releases the lock. Instead, the lock is simply unowned. The scheduler chooses another thread to switch to after A. If it's B, then it gets the lock and continues. If it's instead some thread C which didn't want the lock initially, then C gets to run. If it doesn't request ℓ, then it just computes as normal. If C does request ℓ, it gets it. Maybe it'll release it before B gets its turn, thus enabling more throughput than the fair lock.

One of the ways in which one can then diagnose a lock convoy is to see a lock that has some nonzero number of waiting threads but nobody appears to own it. It just so happens that we're in the middle of a handover; some thread has signalled but the other thread has not yet woken up to run yet.

Changing the locks to be unfair does risk starvation, although one can imagine that it is fairly unlikely given that a particular thread would have to be very low priority and very unlucky. Windows does give a thread priority boost, temporarily, after it gets unblocked, to see to it that the unblocked thread does actually get a chance to run.

Mitigating Lock Convoys Ourselves. Although it can be nice to be able to give away such a problem to the OS developers and say "please solve this, thanks", that might not be realistic and we might have to find a way to work around it. We'll consider four solutions from [Loh05]: Sleep, Share, Cache, and Trylock.

We could make the threads that are NOT in the lock convoy call a sleep() system call fairly regularly to give other threads a chance to run. This solution is lame, though, because we're changing the threads that are not the offenders and it just band-aids the situation so the convoy does not totally trash performance. Still, we are doing a lot of thread switches, which themselves are expensive as outlined above.

The next idea is sharing: can we use a reader-writer lock to allow much more concurrency than we would get if everything used exclusive locking? If there will be a lot of writes then there's limited benefit to this speedup, but if reads are the majority of operations then it is worth doing. We can also try to find a way to break a critical section into two or more smaller ones, if that can be done without any undesirable side effects or race conditions.

The next idea has to do with changing when (and how) you need the data. If you shrink the critical section to just pull a copy of the shared data and operate on the shared data, then it reduces the amount of time that the lock is held and therefore speeds up operations. But you saw the earlier discussion about critical section sizes, right? So you did that already...?

The last solution suggested is to use try-lock primitives: try to acquire the lock, and if you fail, yield the CPU to some other thread and try again. It requires a concept of yielding, of course, and it is fairly straightforward. The `yield_now` function just tells the OS scheduler that we are not able to do anything useful right now and we'd prefer to let another thread run instead. The Rust documentation points out that channels do this in the implementation of sending and receiving to and from channels. But, see the code below for a quick example.

```
let mut retries = 0;
let retries_limit = 10;
let counter = Mutex::new(0);

loop {
    if retries < retries_limit {
        let mut l = counter.try_lock();
```

```
            if l.is_ok() {
                *l.unwrap() = 1;
                break;
            } else {
                retries = retries + 1;
                thread::yield_now();
            }
        } else {
            *counter.lock().unwrap() = 1;
            break;
        }
    }
```

In short, we try to lock the mutex some number of times (up to a maximum of `retries_limit`), releasing the CPU each time if we don't get it, and if we do get it then we can continue. If we reach the limit then we just give up and enter the queue (regular lock statement) so we will wait at that point. You can perhaps think of this as being like waiting for the coffee machine at the office in the early morning. If you go to the coffee machine and find there is a line, you will maybe decide to do something else, and try again in a couple minutes. If you've already tried the come-back-later approach and there is still a line for the coffee machine you might as well get in line.

Why does this work? It looks like polling for the critical section. The limit on the number of tries helps in case the critical section belongs to a low priority thread and we need the current thread to be blocked so the low priority thread can run. Under this scheme, if A is going to release the critical section, B does not immediately become the owner and A may keep running and A might even get the critical section again before B tries again to acquire the lock (and may succeed). Even if the spin limit is as low as 2, this means two threads can recover from contention without creating a convoy [Loh05].

The Thundering Herd Problem. The lock convoy has some similarities with a different problem called the *thundering herd problem*. In the thundering herd problem, some condition is fulfilled (e.g., broadcast on a condition variable) and it triggers a large number of threads to wake up and try to take some action. It is likely they can't all proceed, so some will get blocked and then awoken again all at once in the future. In this case it would be better to wake up one thread at a time instead of all of them.

You may have learned about condition variables earlier. Rust has them as well, the `std::sync::Condvar` type.

The Lost Wakeup Problem. However! Waking up only one thread at a time has its own problems [29]. For instance, on a condition variable you can choose to wake up one waiting thread with either `notify_one()` or all waiting threads with `notify_all()`. If you use `notify_one()`, then you can encounter the *lost wakeup* problem.

The general recommendation of the internet is to use `notify_all` in all situations. Counting on each thread to aways unconditionally wake up the next when it runs is slightly dangerous...

Atomics

What if we could find a way to get rid of locks and waiting altogether? That would avoid the lock convoy problem as well as any potential for deadlock, starvation, et cetera. In previous courses, you have learned about test-and-set operations and possibly compare-and-swap and those are atomic operations supported through hardware. They are uninterruptible and therefore will either completely succeed or not run at all. Is there a way that we could use those sorts of indivisible operations? Yes!

Atomics are a lower-overhead alternative to locks as long as you're doing suitable operations. Remember that what we wanted sometimes with locks and mutexes and all that is that operations are indivisible: an update to a variable doesn't get interfered with by another update. Remember the key idea is: an *atomic operation* is indivisible. Other threads see state before or after the operation; nothing in between.

We are only going to talk about atomics with sequential consistency. That means when you are asked about ordering in a method on an atomic type, it means `Ordering::SeqCst`. Later in the course we will revisit the idea of memory consistency and the different possible reorderings, but for now, just use sequential consistency and you won't get surprises.

[29] `https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again`

So there are atomic types for integer types (signed and unsigned), boolean, size (signed and unsigned), and pointers. It's important to note that when interacting with the type, you cannot just assign or read the value; you're forced to use the `load` and `store` methods to be sure there's no confusion. Such types are safe to be passed between threads as well as being shared between them.

```
use std::sync::atomic::{AtomicBool, Ordering};

fn main() {
   let b = AtomicBool::new(false);
    b.store(true, Ordering::SeqCst);
    println!("{}", b.load(Ordering::SeqCst));
}
```

In addition, there are a few other methods to allow you to atomically complete the operations you normally need. For example, `fetch_add` is what you would use to atomically increase the variable's value. In C, `count++` is not atomic; in Rust we would use `count.fetch_add(1, Ordering::SeqCst)`.

The other atomic operations that we can breeze past are `fetch_sub` (fetch and subtract), `fetch_max` (fetch and return the max of the stored value and the provided argument), `fetch_min` (same as max but minimum), and the bitwise operations `and`, `nand`, `or`, `xor`.

Compare and Swap. This operation is also called **compare and exchange** (implemented by the `cmpxchg` instruction on x86). This is one of the more important atomic operations, because it combines the read, comparison, and write into a single operation. You'll see **cmpxchg** quite frequently in the Linux kernel code.

Here's a description of how a compare-and-swap operation works using C. This is obviously not how it is implemented, but explaining it using program code is more precise (and compact) than a lengthy English-language explanation. It is really implemented as an atomic hardware instruction and this all takes place uninterruptibly.

```
int compare_and_swap (int* reg, int oldval, int newval) {
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

Afterwards, you can check if the CAS returned `oldval`. If it did, you know you changed it. If not, you should try again (maybe with some delay). If multiple threads are trying to do the compare-and-swap operation at the same time, only one will succeed.

The Rust equivalent for this is called `compare_and_swap` and it takes as parameters the expected old value, the desired new value, and the ordering. We'll see an example in just a moment. Rust does offer a simple `swap` on atomic types that doesn't do the comparison and just returns the old value, as well as two more advanced versions called `compare_exchange` and `compare_exchange_weak` that we won't talk about today.

Implementing a Spinlock. You can use compare-and-swap to implement a spinlock. Remember that a spinlock is constantly trying to acquire the lock (here, represented by an atomic boolean) and only makes sense if the expected waiting time to acquire the lock is less than the time it would take for two thread switches.

```
use std::sync::atomic::{AtomicBool, Ordering, spin_loop_hint};

fn main() {
    let my_lock = AtomicBool::new(false);
    // ... Other stuff happens

    while my_lock.compare_and_swap(false, true, Ordering::SeqCst) == true {
        // The lock was ‘true‘, someone else has the lock, so try again
        spin_loop_hint();
    }
    // Inside critical section
    my_lock.store(false, Ordering::SeqCst);
}
```

The call inside the loop to `spin_loop_hint` is just a nicety we can use to tell the CPU that it's okay to either switch

to another thread in hyperthreading or to run in a lower-power mode if we are spinning[30]. Full CPU effort isn't needed for this, and it's nice if we can let the CPU know that.

ABA Problem Sometimes you'll read a location twice. If the value is the same both times, nothing has changed, right? *No.* This is an **ABA problem**.

The ABA problem is not any sort of acronym nor a reference to this [Abb74]. It's a value that is A, then changed to B, then changed back to A. The ABA problem is a big mess for the designer of lock-free Compare-And-Swap routines. This sequence will give some example of how this might happen [DPS10]:

1. P_1 reads A_i from location L_i.
2. P_k interrupts P_1; P_k stores the value B into L_i.
3. P_j stores the value A_i into L_i.
4. P_1 resumes; it executes a false positive CAS.

It's a "false positive" because P_1's compare-and-swap operation succeeds even though the value at L_i has been modified in the meantime. If this doesn't seem like a bad thing, consider this. If you have a data structure that will be accessed by multiple threads, you might be controlling access to it by the compare-and-swap routine. What should happen is the algorithm should keep trying until the data structure in question has not been modified by any other thread in the meantime. But with a false positive we get the impression that things didn't change, even though they really did.

You can combat this by "tagging": modify value with a nonce upon each write. You can also keep the value separately from the nonce; double compare and swap atomically swaps both value and nonce. Java collections do something resembling this. A collection has a modification count and every time the collection is modified in some way (element added, for example) the counter is increased. When an iterator is created to iterate over this collection, the iterator notes down the current value of the modification count. As it iterates over the collection, if the iterator sees that the collection's modification count is no longer the same as the value it has remembered, it will throw a `ConcurrentModificationException`.

Caveats. Obviously, the use of atomic types just ensures that a write or read (or read-modify-write operation) happens atomically; race conditions can still happen if threads are not properly coordinated.

Unfortunately, not every atomic operation is portable. Rust will try its best to give you the atomic types that you ask for. Sometimes emulation is required to make it happen, and an atomic type might be implemented with a larger type (e.g.,, `AtomicI8` will be implemented using a 4-byte type). Some platforms don't have it at all. So code that is focused on portability might have to be a bit careful.

Lock-Freedom

Let's suppose that we want to take this sort of thing up a level: we'd like to operate in a world in which there are no locks. Research has gone into the idea of lock-free data structures. If you have a map, like a HashMap, and it will be shared between threads, the normal thing would be to protect access to the map with a mutex (lock). But what if the data structure was written in such a way that we didn't have to do that? That would be a lock-free data structure.

It's unlikely that you want to use these sorts of things everywhere in your program. For a great many situations, the normal locking and unlocking behaviour is sufficient, provided one avoids the possibility of deadlock by, for example, enforcing lock ordering. We likely want to use it when we need to guarantee that progress is made, or when we really can't use locks (e.g., signal handler), or where a thread dying while holding a lock results in the whole system hanging.

[30] `https://doc.rust-lang.org/std/sync/atomic/fn.spin_loop_hint.html`

Before we get too much farther though we should take a moment to review some definitions. I assume you know what blocking functions are (locking a mutex is one) and that you also have a pretty good idea by now of what is not (spinlock or trylock behaviour).

The definition of a non-blocking data structure is one where none of the operations can result in being blocked. In a language like Java there might be some concurrency-controlled data structures in which locking and unlocking is handled for you, but those can still be blocking. Lock-free data structures are always inherently non-blocking, but that does not go the other way: a spin lock or busy-waiting approach is not lock free, because if the thread holding the lock is suspended then everyone else is stuck [Wil10].

A lock-free data structure doesn't use any locks (duh) but there's also some implication that this is also thread-safe; concurrent access must still result in the correct behaviour, so you can't make all your data structures lock-free ones by just deleting all the mutex code. Lock-free also doesn't mean it's a free-for-all; there can be restrictions, like, for example, a queue that allows one thread to append to the end while another removes from the front, although two removals at the same time might cause a problem.

The actual definition of lock-free is that if any thread performing an operation gets suspended during the operation, then other threads accessing the data structure are still able to complete their tasks. This is distinct from the idea of waiting, though; an operation might still have to wait its turn or might get restarted if it was suspended and when it resumes things have somehow changed. Since we just talked about compare-and-swap, you might have some idea about this already: you try to do the compare-and-swap operation and if you find that someone changed it out from under you, you have to go back and try again. Unfortunately, going back to try again might mean that threads are frequently interrupting each other...

For this you might need wait-free data structures. This does not mean that nothing ever has to wait, but it does mean that each thread trying to perform some operation will complete it within a bounded number of steps regardless of what any other threads do [Wil10]. This means that a compare-and-swap routine as above with infinite retries is not wait free, because a very unlucky thread could potentially take infinite tries before it completes its operations. The wait free data structures tend to be very complicated...

Let's consider some example from [Tur15], with some modifications. We'll start with a lock-free stack.

```
use std::ptr::{self, null_mut};
use std::sync::atomic::{AtomicPtr, Ordering};

pub struct Stack<T> {
    head: AtomicPtr<Node<T>>,
}

struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

impl<T> Stack<T> {
    pub fn new() -> Stack<T> {
        Stack {
            head: AtomicPtr::new(null_mut()),
        }
    }
}

impl<T> Stack<T> {
    pub fn push(&self, t: T) {
        // allocate the node, and immediately turn it into a *mut pointer
        let n = Box::into_raw(Box::new(Node {
            data: t,
            next: null_mut(),
        }));
        loop {
            // snapshot current head
            let head = self.head.load(Ordering::SeqCst);

            // update 'next' pointer with snapshot
            unsafe { (*n).next = head; }

            // if snapshot is still good, link in new node
            if self.head.compare_and_swap(head, n, Ordering::SeqCst) == head {
```

```
                break
            }
        }
    }
}
```

A particularly unlucky thread might spend literally forever spinning around the loop as above, but that's okay because that thread's bad luck is someone else's good luck. At least some thread, somewhere, has succeeded in pushing to the stack, so the system is making progress (stuff is happening).

And here is a small wait-free algorithm:

```
fn increment_counter(ctr: &AtomicI32) {
    ctr.fetch_add(1, Ordering::SeqCst);
}

fn decrement_counter(ctr: &AtomicI32) {
    let old = ctr.fetch_sub(1, Ordering::SeqCst);
    if old == 1 { // We just decremented from 1 to 0
        println!{ "All done." }
    }
}
```

Obviously, the print statement in the decrement counter is just a placeholder for something more useful. Both operations will complete in a bounded number of steps and therefore there is no possibility that anything gets stuck or is forced to repeat itself forever.

The big question is: are lock-free programming techniques somehow better for performance? Well, they can be but they might not be either. Lock-free algorithms are about ensuring there is forward progress in the system and not really specifically about speed. A particular algorithm implementation might be faster under lock-free algorithms. For example, if the compare and swap operation to replace a list head is faster than the mutex lock and unlock, you prefer the lock-free algorithm. But often they are not. In fact, the lock-free algorithms could be slower, in which case you use them because you must, not because it is particularly speedy.

13 — Dependencies and Speculation

Dependencies

Some computations appear to be "inherently sequential". There are plenty of real-life analogies:

- must extract bicycle from garage before closing garage door
- must close washing machine door before starting the cycle
- must be called on before answering questions? (sort of, some people shout out...)
- students must submit assignment before course staff can mark the assignment (also sort of... I can assign you a grade of zero if you didn't submit an assignment!)

There are some prerequisite steps that need to be taken before a given step can take place. The problem is that we need some result or state from an earlier step before we can go on to the next step. Interestingly, in many of the analogies, sometimes if you fail to respect the dependency, nothing physically stops the next step from taking place, but the outcome might not be what you want (...you don't want zero on your assignment, right?).

The same with dependencies in computation. If you need the result of the last step you will have to wait for it to be available before you can go on to the next. And if you jump the gun and try to do it early, you will get the wrong result (if any at all).

Note that, in this lecture, we are going to assume that we don't have data races. This reasoning is not guaranteed to work in the presence of undefined behaviour, which exists when you have data races.

Main Idea. A *dependency* prevents parallelization when the computation XY produces a different result from the computation YX. That is to say, there is a correct order for doing these two steps and getting the order wrong means we get the wrong outcome. If you want to bake a cake, you have to mix all the ingredients before you bake them; if you bake all ingredients and then mix them, whatever you get isn't a cake.

Remember, of course, that there are lots of things that don't have dependencies and can be done in arbitrary order (or even concurrently). If your household chores are to vacuum the floor and do the laundry, you get a valid outcome no matter what order you do them in, as long as you do both correctly. The outcome is the same!

There are two kinds of dependency that we'll cover and they are *loop-carried* and *memory-carried* dependencies.

Loop-Carried Dependencies. Let's start with the loop-carried version. In this kind of dependency, executing an iteration of the loop depends on the result of the previous iteration. Initially, `vec[0]` and `vec[1]` are 1. Can we run these lines in parallel?

```
let mut vec = vec![1; 32];
    /* ... */
vec[4] = vec[0] + 1;
vec[5] = vec[0] + 2;
```

It turns out that there are no dependencies between the two lines. But this is an atypical use of arrays. Let's look at more typical uses.

What about this? (Again, all elements initially 1.)

```
for i in 1 .. vec.len() {
    vec[i] = vec[i-1] + 1;
}
```

Nope! We can unroll the first two iterations:

```
vec[1] = vec[0] + 1;
vec[2] = vec[1] + 1;
```

Depending on the execution order, either `vec[2] = 3` or `vec[2] = 2`. In fact, no out-of-order execution here is safe—statements depend on previous loop iterations, which exemplifies the notion of a *loop-carried dependency*.

Okay, that's perhaps a silly example. Let's try a real problem. Consider this code to compute whether a complex number $x_0 + iy_0$ belongs to the Mandelbrot set.

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
fn mandelbrot(x0: f64, y0: f64) -> i32 {
    let mut iterations = 0;
    let mut x = x0;
    let mut y = y0;
    let mut x_squared = x * x;
    let mut y_squared = y * y;
    while (x_squared + y_squared < 4f64) && (iterations < 1000) {
        y = 2f64 * x * y + y0;
        x = x_squared - y_squared + x0;
        x_squared = x * x;
        y_squared = y * y;
        iterations += 1;
    }
    return iterations;
}
```

In this case, it's impossible to parallelize loop iterations, because each iteration *depends* on the (x, y) values calculated in the previous iteration. For any particular $x_0 + iy_0$, you have to run the loop iterations sequentially.

That doesn't mean that the problem cannot be parallelized at all, however. You can parallelize the Mandelbrot set calculation by computing the result simultaneously over many points at once, even if each point's calculation needs to be done sequentially. Indeed, that is a classic "embarassingly parallel" problem, because the you can compute the result for all of the points simultaneously, with no need to communicate.

Now consider this example—is it parallelizable? (Again, all elements initially 1.)

```
for i in 4 .. vec.len() {
    vec[i] = vec[i-4] + 1;
}
```

Yes, to a degree. We can execute 4 statements in parallel at a time:

- `vec[4] = vec[0] + 1, vec[8] = vec[4] + 1`
- `vec[5] = vec[1] + 1, vec[9] = vec[5] + 1`
- `vec[6] = vec[2] + 1, vec[10] = vec[6] + 1`
- `vec[7] = vec[3] + 1, vec[11] = vec[7] + 1`

We can say that the array accesses have stride 4 — there are no dependencies between adjacent array elements. In general, consider dependencies between iterations.

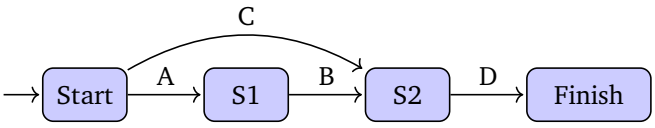
Memory-Carried Dependencies. On the other hand, a memory-carried dependency is one where the result of a computation *depends* on the order in which two memory accesses occur. For instance:

```
let mut acct: Account = Account {
    balance: 0.0f32
};
f(&mut acct);
g(&mut acct);

/* ... */

fn f (a: &mut Account) {
    a.balance += 50.0f32;
}
fn g (a: &mut Account) {
    a.balance *= 1.01f32;
}
```

What are the possible outcomes after executing `g()` and `f()` in parallel threads? Obviously, assuming they are properly wrapped in Atomic reference counted types to allow the code to compile.

Critical Paths

You should be familiar with the concept of a critical path from other courses (e.g., capstone design project) or just from project management you've experienced on coop terms. It is the minimum amount of time to complete the task, taking dependencies into account. Consider the following diagram, which illustrates dependencies between tasks (shown on the arrows). Note that B depends on A, and D depends on B and C, but C does not depend on anything, so it could be done in parallel with everything else. You can also compute expected execution times for different strategies.



Having a diagram like this makes it easy to see what can run in parallel and what steps need to happen in what order. You might not need such a thing, but they are super useful for explaining to nontechnical coworkers or senior management what is happening and how it's going.

Breaking Dependencies with Speculation

Let's go back to a real life analogy of speculation. Under normal circumstances, the coffee shop staff waits for you to place your order ("medium double double") before they start making your order. Sensible. If you go to a certain coffee shop enough, then the staff start to know you and know your typical order and they might speculate about your order and start preparing it in advance, even before you get up to the counter. If they're right, time is saved: your order is ready sooner. If they're wrong, the staff did some unnecessary work and they'll throw away that result and start again with what you did order. If they can predict with high accuracy what you will order, then this is a net savings of time on average.

You can even put some numbers on it! If successfully predicting your order saves 30 seconds per day, and if they're wrong, remaking it takes 3 minutes longer than normal, then if your order is unpredictable one out of every 10 days, how much time is saved? 9 days are correct for a savings of 4.5 minutes, from which we subtract the 3 minute penalty for being wrong, and we've saved 1.5 minutes in the 10 day period (or about 15 seconds per day on average). Not huge, but imagine if the coffee shop can do that for every regular customer. If there are 100 regular customers that's 25 minutes of waiting saved per day! It all adds up.

Mind you, the idea of speculation isn't new in this course. Recall that computer architects often use speculation to predict branch targets: the direction of the branch depends on the condition codes when executing the branch

code. To get around having to wait, the processor speculatively executes one of the branch targets, and cleans up if it has to.

We can also use speculation at a coarser-grained level and speculatively parallelize code. We discuss two ways of doing so: one which we'll call speculative execution, the other value speculation.

Speculative Execution for Threads. The idea here is to start up a thread to compute a result that you may or may not need. Consider the following code:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let val = long_calculation(x, y);
    if val > threshold {
        return val + second_long_calculation(x, y);
    }
    return val;
}
```

Without more information, you don't know whether you'll have to execute `second_long_calculation()` or not; it depends on the return value of `long_calculation()`. Fortunately, the arguments to `second_long_calculation()` do not depend on `long_calculation()`, so we can call it at any point. Here's one way to speculatively thread the work:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let t1 = thread::spawn(move || {
        return long_calculation(x, y);
    });
    let t2 = thread::spawn(move || {
        return second_long_calculation(x, y);
    });
    let val = t1.join().unwrap();
    let v2 = t2.join().unwrap();
    if val > threshold {
        return val + v2;
    }
    return val;
}
```

We now execute both of the calculations in parallel and return the same result as before. Is this the only way to do it? No. The current thread is a valid thread for doing work and we don't have to create two threads and join two threads: we can create one and maybe have less overhead. There may be design reasons for running the work on a separate thread versus the main thread, depending on the situation; for instance, if it's not 2 but 2000 calculations, you probably want the uniformity of running them all on separate threads.

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let t1 = thread::spawn(move || {
        return second_long_calculation(x, y);
    });
    let val = long_calculation(x, y);
    let v2 = t1.join().unwrap();
    if val > threshold {
        return val + v2;
    }
    return val;
}
```

Intuitively: when is this code faster? When is it slower? How could you improve the use of threads?

We can model the above code by estimating the probability p that the second calculation needs to run, the time T_1 that it takes to run `long_calculation`, the time T_2 that it takes to run `second_long_calculation`, and synchronization overhead S. Then the original code takes time

$$T = T_1 + pT_2,$$

while the speculative code takes time

$$T_s = \max(T_1, T_2) + S.$$

Exercise. Symbolically compute when it's profitable to do the speculation as shown above. There are two cases: $T_1 > T_2$ and $T_1 < T_2$. (You can ignore $T_1 = T_2$.)

Also, if you have more cores than you need, maybe wasted work only costs you a bit of extra power.

Value Speculation. The other kind of speculation is value speculation. In this case, there is a (true) dependency between the result of a computation and its successor:

```
fn do_other_work(x: i32, y: i32) -> i32 {
    let val = long_calculation(x, y);
    return second_long_calculation(val);
}
```

If the result of `value` is predictable, then we can speculatively execute `second_long_calculation` based on the predicted value. (Most values in programs are indeed predictable, just like how most branches are predictable). If 90% of customers are using the standard billing plan, you might assume that that's correct in your calculations, and then change that later if it turns out you are wrong. Or you might have a software cache where you keep the latest state so you don't have to go to the database to check a value that changes rarely.

```
fn do_other_work(x: i32, y: i32, last_value: i32) -> i32 {
    let t = thread::spawn(move || {
        return second_long_calculation(last_value);
    });
    let val = long_calculation(x, y);
    let v2 = t.join().unwrap();
    if val == last_value {
        return v2;
    }
    return second_long_calculation(val);
}
```

Note that this is somewhat similar to memoization, except with parallelization thrown in. In this case, the original running time is

$$T = T_1 + T_2,$$

while the speculatively parallelized code takes time

$$T_s = \max(T_1, T_2) + S + pT_2,$$

where S is still the synchronization overhead, and p is the probability that `val != last_value`.

Exercise. Do the same computation as for speculative execution.

When can we speculate?

Speculation isn't always safe. We need the following conditions:

- `long_calculation` and `second_long_calculation` must not call each other.
- `second_long_calculation` must not depend on any values set or modified by `long_calculation`.
- The return value of `long_calculation` must be deterministic.

As a general warning: Consider the *side effects* of function calls. Oh, let's talk about side effects. Why not. They have a big impact on parallelism. Side effects are problematic, but why? For one thing they're kind of unpredictable (why does calling this function result in unexpected changes elsewhere?!). Side effects are changes in state that do not depend on the function input. Calling a function or expression has a side effect if it has some visible effect on the outside world. Some things necessarily have side effects, like printing to the console. Others are side effects which may be avoidable if we can help it, like modifying a global variable. As we've seen, Rust discourages those

kinds of problems but doesn't forbid them: we can still have atomic types shared that would represent some sort of global state.

Software Transactional Memory

Somewhat related to the idea of speculation is the idea of software transactional memory. In this case, a group of changes are carried out on a speculative basis, assuming that they will succeed, and we'll check afterwards if everything is okay and retry if necessary [ST95].

There is a library for this in Rust[31] although we can't vouch for its quality or usefulness. Developers use software transactions by writing `atomic` blocks:

```
let x = atomically(|trans| {
    var.write(trans, 42)?; // Pass failure to parent.
    var.read(trans) // Return the value saved in var.
});
```

The idea resembles database transactions, which most likely you know about. The `atomic` construct means that either the code in the atomic block executes completely, or aborts/rolls back in the event of a conflict with another transaction (which triggers a retry later on, and repeated retries if necessary to get it applied).

Benefit. The big win from transactional memory is the simple programming model. It is far easier to program with transactions than with locks. Just stick everything in an atomic block and hope the compiler does the right thing with respect to optimizing the code.

Motivating Example. We'll illustrate STM with the usual bank account example[32].

```
struct Account {
    balance: TVar<f32>,
}

fn transfer_funds(sender: &mut Account, receiver: &mut Account, amount: f32) {
    atomically(|tx| {
        let sender_balance = sender.balance.read(tx)?;
        let receiver_balance = receiver.balance.read(tx)?;
        sender.balance.write(tx, sender_balance - amount)?;
        receiver.balance.write(tx, receiver_balance + amount)?;
        Ok(0)
    });
}
```

Using locks, we have two main options:

- Big Global Lock: Lock everything to do with modifying accounts. This is slow and serializes your program.
- Use a different lock for every account. Prone to deadlocks; high overhead.

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

Drawbacks. As I understand it, three of the problems with transactions are as follows:

- I/O: Rollback is key. The problem with transactions and I/O is not really possible to rollback. (How do you rollback a write to the screen, or to the network?)
- Nested transactions: The concept of nesting transactions is easy to understand. The problem is: what do you do when you commit the inner transaction but abort the nested transaction? The clean transactional façade doesn't work anymore in the presence of nested transactions. (The Rust library will panic at runtime if you try to nest transactions)

[31] `https://github.com/Marthog/rust-stm`

[32] It turns out that bank account transactions aren't actually atomic, but they still make a good example.

- Transaction size: Some transaction implementations (like all-hardware implementations) have size limits for their transactions.

Implementations. Transaction implementations are typically optimistic; they assume that the transaction is going to succeed, buffering the changes that they are carrying out, and rolling back the changes if necessary.

One way of implementing transactions is by using hardware support, especially the cache hardware. Briefly, you use the caches to store changes that haven't yet been committed. Hardware-only transaction implementations often have maximum-transaction-size limits, which are bad for programmability, and combining hardware and software approaches can help avoid that.

Implementation issues. Since atomic sections don't protect against data races, but just rollback to recover, a datarace may still trigger problems in your program.

```
fn what_could_go_wrong(x: TVar<i32>, y: TVar<i32>) {
    atomically(|t| {
        let old_x = x.read(t)?;
        let old_y = y.read(t)?;
        x.write(t, old_x + 1);
        y.write(t, old_y + 1);
        Ok(0)
    });
}
```

```
fn oh_no(x: TVar<i32>, y: TVar<i32>) {
    atomically(|transaction| {
        if x.read(transaction)? != y.read(transaction)? {
            loop { /* Cursed Thread */}
        }
        Ok(0)
    });
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can. That's because intermediate states can still become visible! Although the block is atomic, that just means the changes succeed or are rolled back as a group; it does not mean that another thread cannot read `x` and `y` and see them at some partial-completion state of the transaction. (Maybe this doesn't happen in the Rust implementation, but it certainly can in C/C++ versions of STM.)

14— Early Termination, Reduced-Resource Computation

Trading Accuracy for Time

Knowing when to quit is wise. In some cases, we can speed up our program by not waiting for the slowest steps to be done. This is somewhat related to speculation, but the big distinction is that in speculation we do extra work "just in case" and with early phase termination, we skip doing some work even though we're supposed to do on the basis of "close enough is good enough". There are two basic ideas: the first way is to skip some parts of work and the second is to intentionally reduce accuracy to speed things up.

You may implement these strategies when you're writing an exam: time is limited and you might choose not to do a certain question because the benefit is small and you can use your time better doing a different question. In which case you might leave question 3.2 blank in favour of working on question 4.1. That's where you skip some work. Alternatively, you could choose to skip error handling in question 4.1, knowing that you will lose some marks in that question but freeing up some more time to do question 3.2. Exams are nice (or nasty) in that we can do both things, but your program might support only one.

Early Phase Termination

A formal name for the first idea, quitting early, is early phase termination [Rin07]. So, to apply it to a concrete idea: we've talked about barriers quite a bit. Recall that the idea is that no thread may proceed past a barrier until all of the threads reach the barrier. Waiting for other threads causes delays. Killing slow threads obviously speeds up the program. Well, that's easy.

"Oh no, that's going to change the meaning of the program!"

Let's consider some arguments about when it may be acceptable to just kill (discard) tasks. Since we're not completely crazy, we can develop a statistical model of the program behaviour, and make sure that the tasks we kill don't introduce unacceptable distortions. Then when we run the program, we get an output and a confidence interval.

If you wanted a game-relevant example, pretend you're really bad at Mario Kart. If you're in last place when the second-last player (or AI) drives across the finish line, the race is over at that point because we already know you finished last ("Oh nooo!"). There's no benefit to waiting while you have to drive the rest of the lap to the finish. In that case, ending the race while one driver has not yet finished is perfectly safe because the outcome is already known: I'm really bad at Mario Kart.

Should Have Made A Left Turn At Albuquerque. Many problems are mathematically hard in nature: to find the optimal solution you have to consider every possibility. Well, what this strategy presupposes is: don't. Imagine the travelling salesperson problem, just for the sake of an example. There are n points to visit and you want to minimize the amount of travel time. The only way to know if a solution is best is to consider every possible route.

One way we can know if we're wasting time is to remember previous outcomes. The solution we're evaluating will

have some travel cost in units (maybe kms). If the currently-accumulated cost in kms is larger than the total of the thus-far best solution, give up. To be specific, if we have a route that has 400 km of driving and we are partway through building a solution and we have already got 412 km of driving, we can give up on this option (and not evaluate the rest of it) because we already know it won't be the best.

Another approach is to stop as soon as you have a solution that's reasonable. If our target is to get total travel under 500 km then we can stop searching as soon as we find one that satisfies this constraint. Yes, we might stop at 499 km and the optimal solution might be 400 (25% more driving for the poor peon)—but it does not have to be perfect; it just has to be acceptable. And if traffic in the hypothetical region is anything like that of the GTA, the route that is shortest in kilometres may not be the shortest in terms of time anyway.

You can also choose to reduce the amount of effort by trying, say, five or ten different possibilities and seeing which of those is the best. There's no guarantee you'll get an optimal solution: you might have randomly chosen the ten worst options you could choose.

Interesting to think about: what does Google Maps do? For some problems there are relatively few solutions; if you plan to drive in the Yukon territory there are a finite set of roads to travel. But suppose you're driving around Toronto; the grid system means there are lots and lots of options, right? Maybe some heuristic is used to generate some possibilities and the best ones of those are chosen.

This Point is Too Hard. Monte Carlo simulations are a good candidate; you're already picking points randomly. Raytracers can work as well. Both of these examples could spawn a lot of threads and wait for all threads to complete. For mathematical functions that are "not nice", different points might take longer to evaluate than others. In either case, you can compensate for missing data points, assuming that they look similar to the ones that you did compute. If you have a function where some graph is being computed, you can probably guess that a missing point is somewhere in between the two (or n) nearest points. So just average them.

The same is true for graphics, of course: if rendering a particular pixel did not go well for some reason, you can just average the adjacent ones and probably people would not notice the difference. Not bad!

In other cases, some threads simply take too long, but we don't need all of them to produce a result. If we are evaluating some protocol where the majority wins, we can stop as soon as sufficient results have been returned; either an outright majority for an option or that the remaining votes couldn't change the outcome. This happens to some extent with election projections: even if not all polling stations are reporting a result, news channels will declare a winner if the remaining votes would not be enough to change the outcome. Actually, news channels probably take it a bit too far in that they will declare a winner even if the outstanding votes exceed the margin, on a theory that it probably won't be the case that they are 100% for the candidate who is in second place. But they can be wrong.

Slow Road... For some categories of problem, we know not only that a solution will exist, but also how many steps it takes to solve (optimally). Consider the Rubik's Cube—it's much easier to explain if you have seen one. It'll appear in the slides, but if you're just reading the note(book/s) then I suggest you google it[33].

This is a problem with a huge number of possible permutations and brute force isn't going to work. However, research has proven that no matter what the state of the cube is, it can be transitioned to a solved state in 20 moves or fewer. This number is called God's Number, presumably because it is the maximum number of moves it would take an all-knowing deity to solve the puzzle. So if you have a solver for a Rubik's cube, and if you don't find a solution in (fewer than) 20 moves, you should cancel this solution attempt and try another one.

Okay, that's fun to talk about, but it's always better if we see it in action? Let's play around with `https://rubiks-cube-solver.com/`, which implements this very behaviour. It says in their description of how it works that it runs an open source algorithm; it looks for a solution in 20 steps or fewer. This implementation does both kinds of tradeoff: if the solution being evaluated takes too long it's killed. And if no under-20-move solution has been found within a certain time limit, it will return a solution that takes 24 steps and give you a less optimal solution. That's actually an example of reducing accuracy (quality of solution) for speed, which leads us into our next approach.

[33] If you've waited for the exam to read this and you can't google... whoops!

Reduced-Resource Computation

The formal name for the second idea is "reduced resource computation"—that is to say, we do more with less! Austerity programs for our computer programs. Well, you can use `float` instead of `double`. Indeed, Google's Tensor Processing Units appear to use perhaps even fewer than 8 bits to represent a floating-point number. Or, you can also work with integers to represent floating point numbers (e.g., representing money in an integer number of cents). But let's really think about when this is appropriate.

Circuit... Analysis! Recall that, in scientific computations, you're entering points that were measured (with some error) and that you're computing using machine numbers (also with some error). Computers are only providing simulations, not the ground truth; the question is whether the simulation is good enough.

Imagine that the simulation is deciding on what resistors are going to be put in your circuit board: is there any point in calculating it down to five decimal places when the resistors you buy have a tolerance of $\pm 5\%$? No, and if you took a circuits course with Prof. Barby he would be very disappointed if you said yes.

iddqd. Perhaps my favourite example of trading accuracy for time is a function in Quake III, contributed by John Carmack known as "fast inverse square root". For graphics processing, sometimes you want to calculate $1/\sqrt{x}$. This is important because you use it in calculating lighting and reflections (because you normalize vectors). Normalizing is mostly a straightforward exercise: square some numbers, add them up, and then... oh no, you have to use square root... That one isn't so simple.

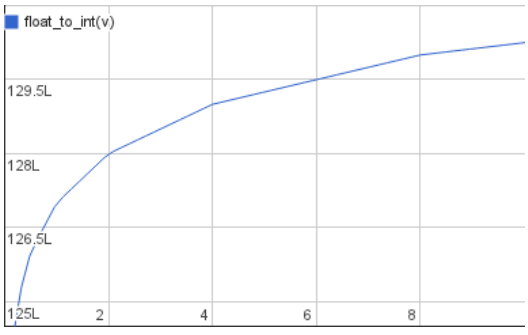
Square root (or similar) is usually calculated by some interpolation or root-finding method (if you took a numerical methods course, you know several techniques for calculating this). But instead there's this [Han12a].

```
float FastInvSqrt(float x) {
  float xhalf = 0.5f * x;
  int i = *(int*)&x;         // evil floating point bit level hacking
  i = 0x5f3759df - (i >> 1);  // what the fuck?
  x = *(float*)&i;
  x = x*(1.5f-(xhalf*x*x));
  return x;
}
```

The first line of the function is straightforward - take half the value of `x`. The second one says to interpret the value of `x` as an `int`. Now this probably seems like dark magic, and it is. Pretend this floating point number is an integer. I mean, you can, but why does this make sense?

There's a lot of explanation and a lot of math in the source material but it comes down to how the float is stored. The float starts with a sign bit, then the exponent, and then the mantissa (math reminder: in 1.95×10^3, the exponent is 3 and the mantissa is 1.95).

The clever hack is somewhat obsoleted now by the fact that CPU instructions now exist to give you fast inverse square root. This was obviously not something you could rely on in 1999 (and is pretty CISC-y), but we're going to revisit the idea of using clever CPU instructions to speed things along in the next lecture. So if we say pretend this float is an integer we end up with this [Han12b]:



If it's not obvious, this plot rather resembles the plot of $-1/\sqrt{x}$. So we are pretty close to getting where we need to go. All we need is to invert it and then do a little bit of an offset. The seemingly magic number of `0x5f3759df`

is not a bit pattern, but just a calculated offset to make the approximation a little bit better. Then we turn it back into a float.

The last step is then to do one quick iteration of Newton's method to refine the calculation a little bit and we have a great solution: it is a fast, constant-time calculation for something that normally would be difficult, and it's very accurate, something like 0.175% error at the most. And in a 3D game a tiny inaccuracy is not a big deal! Especially in one from 1999. It wasn't exactly photorealistic to begin with, now was it...?

This is the best case scenario: the accuracy that we trade for speed is both very small and its application is one in which a small difference is not noticeable. This is beyond "close enough is good enough", this is hardly any tradeoff at all.

N-Body Problem A common physics problem that programmers are asked to simulate is the N-Body problem: you have some number of bodies (N, obviously) and they interact via gravitational forces. The program needs to compute the movements of the bodies over time. This is a typical example of a program that is well suited to parallelization: you can compute the forces on each body n from all other bodies in parallel. This was even at one time an OpenCL assignment in this course, although now there are too many good solutions on the internet so it was replaced. Bummer.

What can you do here if you want to speed it up even more? You could look for optimizations that trade off accuracy for performance. As you might imagine, using `float` instead of `double` can save half the space which should make things quite a bit faster. But you want more...

Then we need some domain knowledge. That is, we need to think about what we know about the problem and we can make a decision about what is important and what is not. If we thought about what's important for determining the forces, what would we consider to be the most important?

Hint: consider the formula: $F = \frac{Gm_1m_2}{r^2}$.

Force is a function of how close the objects are. Thus, points that are far away contribute only small forces. So you can estimate them (crudely). A first approximation might say that forces that are far enough away are zero. In principle, Jupiter has a gravitational influence on the rate of descent if I drop a whiteboard marker (whether positive, negative, or zero depends on its relative position at the time of my clumsiness), but the effect is so incredibly small as to be worth ignoring. But what about objects that are not exactly close by, but also not so far away as to be irrelevant?

The idea is to divide the points into a number of "bins" which are cubes representing a locale of some sort. Then, compute the centre of mass for each bin. When calculating the forces on a given point, add the force exerted by the centre of mass for faraway bins to the force exerted by individual particles for nearby particles.

A more concrete explanation with an example: suppose the space is divided into $[0, 1000]^3$, so we can take bins which are cubes of length 100. This gives 1000 bins. If you want to increase the accuracy, increase the number of bins. If you want to increase the speed, decrease the number of bins: either make bins larger, or change your definition of what is too far away to care about.

The program should have a 3-dimensional array `cm` of a point structure to store centres-of-mass. The `x`, `y` and `z` components contain the average position of the centres of mass of a bin, while the `mass` component stores the total mass. Compute all of the masses in parallel: create one thread per bin, and add a point's position if it belongs to the bin, e.g.

```
struct Point {
    x: f32,
    y: f32,
    z: f32,
    mass: f32,
}
```

Let's start there. We are going to improve this by adding a `bin` property to each point, so that we know what bin it is in. Later, we can use the bin to know if another point is considered close by. In my example, I calculate the bin at the same time as the point is randomly generated, because why iterate over the collection a second time?

Once all points are generated, we can calculate the centre of mass for each bin. This is, of course, just a weighted average of all the points in that bin and is straightforward to calculate.

The payoff from all these calculations is to save time while calculating forces. In this example, we'll compute exact forces for the points in the same bin and the directly-adjacent bins in each direction (think of a Rubik's Cube; that makes 27 bins in all, with 6 bins sharing a square, 12 bins sharing an edge, and 8 bins sharing a vertex with the centre bin). If there is no adjacent bin (i.e., this is an edge), just act as if there are no points in the place where the nonexistent bin would be.

This does mean there is overhead for each step, meaning the total amount of overhead goes up. We had to (1) calculate what bin this is, (2) calculate the centre of mass for each bin, and (3) decide when we should use the centre-of-mass calculation or the exact calculation.

Here's some data calculated with 100 000 points (using `hyperfine -m 5 "cargo run -release"`). The unmodified version takes about 162 seconds; the modified version takes about 147. With smaller numbers of points, the difference is not as noticeable, but still consistent. With 50 000 the original `nbody` program takes about 39 seconds on average and the optimized about 37, so a slight speedup! The amount of benefit increases with more points, but doesn't keep up with the computational complexity of the increase in the number of points.

Also, this is before any parallelization (no threads are spawned). We can calculate forces on each point pretty effectively in parallel; we can also parallelize the calculations of the centre of mass quite easily. Both would speed up the program quite a lot!

If I just parallelize the version without approximations (using the rayon parallel iterator), it takes about 25 seconds to run, and parallelizing the version with bins (using the same in a very naive parallelization) gets the execution time for 100 000 points down to about the same 25 seconds. It is clear that parallelizing the problem has a much greater effect than the tradeoff of accuracy for time (at least in this implementation), but on a sufficiently large problem, everything counts.

Loop perforation

You can also apply the same idea to sequential programs. Instead of discarding tasks, the idea here is to discard loop iterations [HMS+09]. Here's a simple example: instead of the loop,

```
for i in 0 .. n { sum += numbers.get(i).unwrap(); }
```

simply write,

```
for i in (0 .. n).step_by(2) { sum += numbers.get(i).unwrap(); }
```

and multiply the end result by a factor of 2. This only works if the inputs are appropriately distributed, but it does give a factor 2 speedup.

Example domains. In [RHMS10], we can read that loop perforation works for evaluating forces on water molecules (in particular, summing numbers); Monte-Carlo simulation for swaption pricing; and video encoding. In that example, changing loop increments from 4 to 8 gives a speedup of 1.67, a signal to noise ratio decrease of 0.87%, and a bitrate increase of 18.47%, producing visually indistinguishable results.

15 — Memory Consistency

Memory Consistency, Memory Barriers, and Reordering

Previously, when atomics were introduced, we said to use sequential consistency without much detail and without discussing the other options. Now it's time to learn about it. We'll cover both instruction reordering by the CPU and reordering initiated by the compiler.

Compiler Reordering. When asked to compile code, the compiler does not take every statement that you provide and translate it into a (set of) machine language instruction(s). The compiler can change the order of certain events. The compiler will be aware of things like load-delay slots and can swap the order of instructions to make use of those slots more effectively. In the (silly) example on the left there might be a stall while we wait for `x` to be available before we can send it in to the `println!` macro; on the right we moved two unrelated instructions into the delay slots. So that feels like free performance!

```
let x = thing.y;
println!("x␣=␣{}", x);
z = z + 1;
a = b + c;
```

```
let x = thing.y;
z = z + 1;
a = b + c;
println!("x␣=␣{}", x);
```

By the way, if you have any undefined behaviour in your program, the compiler is allowed to do anything it wants when it reorders or otherwise optimizes it; maybe you can see why that makes sense. We'll talk about other compiler optimizations soon, but we don't want to get away from the topic of reordering.

Hardware Reordering. In addition to the compiler reordering, the hardware can do some reordering of its own. A sequence of instructions is provided to the CPU, and it can decide it would rather do them in an order it finds more convenient. That is fairly straightforward.

There is another possibility we have to consider, and it is updates from other threads. When a thread is doing a check on a variable, such as a quit condition (exit the loop if `quit` is now true), how do we know if we have the most up-to-date value for `quit`? We know from the discussion of cache coherence that the cache will be updated via snooping, but we need a bit more reassurance that the value we're seeing is the latest one. How could we get the wrong order? If the read by thread A is reordered by the hardware so that it's after the write by thread B, then we'll see the "wrong" answer.

Different hardware provides different guarantees about what reorderings it won't do. Old 386 CPUs didn't do any; modern x86 usually won't (except where there are some specific violations of that); ARM has weak ordering except where there are data dependencies [Pre12]. ARM is getting pretty popular, so we do have to care about hardware reorderings, unfortunately.

I have a plan, but it's a bad one. There are some reorderings where we are easily able to conclude that it is okay and safe to do, but not every reordering is. In an obvious case, if the lines of code are `z *= 2` and `z += 1` then neither the compiler nor hardware will reorder those because it knows that it would change the outcome and produce the wrong answer. There's a clear data dependency there, so the reordering won't happen. There are a couple of hardware architectures where that isn't respected, but we'll ignore them for now.

But what if there's no such clear dependency? Consider something like this pseudocode:

```
lock mutex for point                    lock mutex for point
point.x = 42;                           point.x = 42;
point.y = -42;                          point.y = -42;
point.z = 0;                            unlock mutex for point
unlock mutex for point                  point.z = 0;
```

Wait a minute—that's not an okay reordering, because now an element of the point is being accessed outside of the critical section and we don't want that. (Not undefined behaviour!) It's a reordering, alright, in that the store of `point.z` has been moved to after the store of state of the mutex (unlock does, after all, change its state). What we need is a way to tell the compiler (and hardware) that this is not okay.

Sequential Consistency. In a sequential program, you expect things to happen in the order that you wrote them. So, consider this code, where variables are initialized to 0:

```
T1: x = 1; r1 = y;
T2: y = 1; r2 = x;
```

We would expect that we would always query the memory and get a state where some subset of these partially-ordered statements would have executed. This is the *sequentially consistent* memory model. A simple description: (1) each thread induces an *execution trace*; and (2) always, the program has executed some prefix of each thread's trace. Or, alternatively:

> "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." — Leslie Lamport

It turns out that sequential consistency is expensive to implement. Think how much coordination is needed to get a few people to agree on where to go for lunch; now try to get a group of people to agree on what order things happened in. Right. Now imagine it's a disagreement between threads so they don't have the ability to negotiate. So most systems actually implement weaker memory models, such that both `r1` and `r2` might end up unchanged.

Allowing some reorderings could potentially significantly speed up the program! If left to its own devices, the compiler could reorder anything, but we need to tell it what is allowed and what is disallowed.

Memory Consistency Models

Rust uses the same memory consistency models as C++. The Rustonomicon (book of names of Rust[34]) says pretty directly that this is not because the model is easy to understand, but because it's the best attempt we have at modelling atomics because it is a very difficult subject. The idea behind the memory model is to have a good way of talking about the *causality* of the program. While causality definitely sounds like something Commander La Forge would talk about on the *Enterprise*, in this case it means establishing relationships between events such as "event A happens before event B".

You will recall from the introduction to the subject of concurrency that we frequently sought the same thing in our program at a higher level, when we'd say that we can use a semaphore to ensure that one thing happens before another. The idea is the same, but our toolkit is a little bit different: it's the *memory barrier* or *fence*.

This type of barrier prevents reordering, or, equivalently, ensures that memory operations become visible in the right order. A memory barrier ensures that no access occurring after the barrier becomes visible to the system, or takes effect, until after all accesses before the barrier become visible.

The x86 architecture defines the following types of memory barriers:

[34] Not to be confused with the Necronomicon...

- `mfence`. All loads and stores before the barrier become visible before any loads and stores after the barrier become visible.
- `sfence`. All stores before the barrier become visible before all stores after the barrier become visible.
- `lfence`. All loads before the barrier become visible before all loads after the barrier become visible.

Note, however, that while an `sfence` makes the stores visible, another CPU will have to execute an `lfence` or `mfence` to read the stores in the right order.

Consider the example again:

```
                        f = 0

/* thread 1 */                      /* thread 2 */
while (f == 0) /* spin */;          x = 42;
// memory fence                     // memory fence
printf("%d", x);                    f = 1;
```

This now prevents reordering, and we get the expected result.

Memory fences are costly in performance. It makes sense when we think about it, since it (1) prevents re-orderings that would otherwise speed up the program; and (2) can force a thread to wait for another one. Sequential consistency will necessarily result in memory fences being generated to produce the correct results.

Other Orderings

The C++ standard includes a few other orderings that don't appear in this section because they aren't in Rust. But we'll cover Acquire-Release and Relaxed briefly. Neither comes with a recommendation to use it, but if you can prove that your use of it is correct, then you can do it. It may give a slight performance edge.

Acquire means that accesses (reads or writes) after the acquire operation can't move to be before the acquire. Release means accesses before the release operation can't move to be after the release. They make a good team: by placing acquire at the start of a section and release after, anything in there is "trapped" and can't get out.

That makes them the perfect combination for a critical section: acquire prevents things from moving from inside the critical section to before the critical section; release prevents things from inside from moving to after the critical section. Nice!

Here's an example of acquire and release, as taken from the Rustonomicon's page about atomics (`https://doc.rust-lang.org/nomicon/atomics.html`. It's implementing a spinlock:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    let lock = Arc::new(AtomicBool::new(false)); // value answers "am I locked?"

    // ... distribute lock to threads somehow ...

    // Try to acquire the lock by setting it to true
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // broke out of the loop, so we successfully acquired the lock!

    // ... scary data accesses ...

    // ok we're done, release the lock
    lock.store(false, Ordering::Release);
}
```

The acquire and release semantics keep all the things that should be in the critical section inside it.

And then there is relaxed. Relaxed really does mean the compiler will take it easy, and all reorderings are possible. Even ones that you might not want! The Rustonomicon suggests one possible valid use for that scenario is a counter

that simply adds and you aren't using the counter to synchronize any action. Something like atomically counting the requests to each resource might be suitable. You can report the counters as metrics and it's not super important that request 9591's increment of the counter occurs before that of request 9598. It's all the same in the end...

Matters, Order Does

There have been a few reminders to use sequential consistency because atomics are hard to reason about and it's easy to get it wrong. But does this happen in reality? Yes, and here's an example of it in Rust [O'C18].

The observed behaviour was an inconsistent state being reported by an assertion; when looking at the registers the registers contained garbage even though it was just after a read that should have loaded it in. That's hard to notice and difficult to debug as well, because running in debug mode might prevent the reordering in the first place.

You can actually look at the fix applied to the lock-free queue at `https://github.com/crossbeam-rs/crossbeam/pull/98/files`. But the short summary is that the load of the `ready` property needs to have at least Acquire semantics and the store of it should have release. If we don't do that, we might attempt to park the thread early.

16 — Rate Limits

It's Not Me, It's You

Sometimes the limiting factor in our application is something that's not under our control. Whenever your application connects to some other application (whether external or not), we may run up against a rate limit.

A *rate limit* represents the maximum rate at which a requester can make requests of the service—it is exactly what it sounds like. In other words, how many requests can be submitted in a given unit of time? Requests above the limit are rejected. If we are making HTTP requests, we are supposed to get a response code of 429 for a request rejected due to a rate limit.

Rate limits can be more complicated than just a simple measurement of requests per unit time. For example, there can be multiple thresholds (e.g., A requests per hour or B requests per day), and it may be segmented by request type/responses (e.g., max C requests to change your user data per day, and a max of D requests of any type per day in total). Responses can be more complicated than just rejection also—the service could intentionally delay or deprioritize requests that exceed a threshold—but for our discussion we'll just stick with simple rejection.

Rejected requests can be a huge problem for your application or service. An obvious example here is something like ChatGPT: if you're using that as a service in your application and you run up against the rate limit for that, some aspect of your application (maybe the critical one?) is unavailable or not working as well as expected. I [JZ] have even got a fun story about getting rate limited by a payment processing platform because they would (1) generate an invoice and notify a service via a webhook, (2) our service would then validate the webhook notification is valid by calling the platform, and (3) our service would then get rate limit errors saying it is calling the validation endpoint too often in a short period of time. Wait, they're calling us and we're just checking that the call is valid—but unfortunately, they don't care, HTTP 429—Too Many Requests!

Why is this happening to me? Rate limits exist because every request has a certain cost associated with it. It takes work, however small or large this might be, to respond to the request. The cost may or may not be measured in a currency—if you are paying for CPU time at a cloud provider, then it literally is measured in monetary units—but it can also be measured in opportunity cost. What do I mean? If the system is busy, responding to one request may mean a delay in responding to other requests. If some of those requests are fraudulent or otherwise invalid, it's taking away time or resources from other, legitimate requests. If the system is overprovisioned, rate limits aren't necessary, but here we're talking about systems that are running closer to the edge.

Denial-of-Service (DoS) attacks are ways that attackers can negatively impact the functioning of a service by simply sending in many invalid requests. The regular DoS attack becomes a Distributed Denial of Service (DDoS) attack when the attacker sends in those requests from numerous clients. This not only allows the attacker much more capacity to send in requests, but also prevents simple solutions like blocking the IP of the offending system. Given enough invalid requests, it can overwhelm the service, which will either cause it to crash, exhaust its resources (e.g., network handles), or be so slow as to be unusable.

An example of a rate limit that I [PL] encountered was in using a system that allowed Ontarians to write letters to the Minister of the Environment about allowing rock climbing in provincial parks. We were encouraging people at climbing gyms to send letters (well, emails), but then the climbing gym IP address got rate limited because people were sending letters from the gym's wifi. We never did get a satisfactory resolution from the letter writing service.

(I'd claim this is also an example of knowing the local context: we think that Canadians have lower data limits[35] than Americans, and thus more likely to be on public wifi rather than just sending from their own connections; the letter writing service didn't consider that local context.)

In another illustration of how requests can take up resources, we can also consider a proposed, and later retracted, approach that the Unity engine wanted to take in 2023 [Bat23]. Unity makes a game engine that people use to make fun games. The company's licensing model didn't really take into account the Live-Service kind of game (think Fortnite or Diablo 4 or similar) where the game is "free to play" but the company that makes it gets the money though selling cosmetic items, power boosts, or just generally preying on people with gambling addictions (via lootboxes which are legally considered gambling in some countries). Unity doesn't get a cut of that, so they wanted money and they wanted to charge per installation. It sounds reasonable, but of course people immediately realized that if you don't like the company who made a game, you could bankrupt them by just repeatedly installing and uninstalling the game. Even if it's only one cent at a time, automate it and do this enough times and you are a major line item on the balance sheet. Unity walked it back and said they want it to be first-install but per device, but a clever enough person would have no trouble making each install seem like it's a different machine (just lie to the application about hardware IDs or something). Suffice it to say, game developers using the engine revolted and some of them decided they might like to switch to a different engine. So if every request to your service is costing you, there's got to be a way to control that cost.

Even if it's not about preventing a DoS attack or monetary costs, rate limits also prevent someone from scraping all your data. This really happened in early 2021 to the openly-political social media site "Parler". I would certainly forgive you for not being familiar with the site since it was only popular amongst conservative Americans and mostly in the year 2020 before all major companies refused to do business with them. But in any case, according to [Gre21] it was trivially easy for people to download all the content from the site because of two very bad decisions. The first was that all posts had sequential identifiers, so it is trivial to just enumerate all posts (don't do this!). The second is that there was no rate limiting so a single person could then get all the data with a simple script. Okay, you might not be so worried about URLs being completely predictable because that is a very silly thing to implement. Still, it is concerning if others can scrape all your content for their own benefit at your cost, even if it's intended as non-malicious training some machine learning model.

The reference to training the machine learning model is an oblique reference to ChatGPT and how it got its data set. Yes, OpenAI did scan the internet to train ChatGPT, although probably not as invasively as how people were pulling the data out of Parler. We can discuss whether it's fair or reasonable for OpenAI to request large amounts of data from others (getting each site to serve up their data at their own execution costs) for OpenAI to then monetize in their model, but that's for another course. Ethics around AI is a big topic indeed—one that you will likely have to wrestle with in your careers.

Dealing With Limits

OK, let's assume that some system you're dealing with has a rate limit, and that we need to address this limitation if we want to increase the performance of the code that is under our control. If the rate limit is super high as compared to our usage of it (e.g., you can do 1000 requests per hour and you are sending 10) then there's no problem right now, but maybe there will be in the future. Probably that is a problem best handled by Future You. Hitting the rate limit regularly, however, is not only frustrating for normal workflows we're trying to run, but could also get us banned as a customer, if it happens too often.

If we start seeing requests rejected with an error that says the rate limit is exceeded, then we are certain it's a problem. It's not always obvious what the limit is, though.

In some scenarios, the documentation tells you the limit, or perhaps there's an API to query it, or API responses for other requests include information about the remaining capacity available to you.

Sometimes, though, the other service does not publish or give information about the rate limit, fearing that it might be exploited or encourage bad behaviour, or because they don't want to commit to anything. Atlassian (they make JIRA, if you're familiar with that) says "REST API rate limits are not published because the computation logic is evolving continuously to maximize reliability and performance for customers" which feels a little bit like they don't want to commit to specific numbers because then someone might be mad if those numbers aren't met. There is an argument that it does allow more freedom to change the API if there's no commitment to a specific rate.

[35] It seems like the federal government has recently fixed this! But it was true at time of writing.

(Unpublished rate limits were in place for the letter-writing campaign service). Testing for the limit (spam until we get rejections) might work, but it might also get you banned and, given the above, may give you a number that quickly becomes out of date.

This section would be short indeed if the answer is to shrug our shoulders and say there's nothing we can do. Of course there is! And yet, when I [JZ] was conducting a system design interview for a candidate in 2022, I presented with a scenario with a rate limit. I got the answer that there was nothing that we could do to solve it. That's defeatist and false, as evidenced by the remainder of this section. As you may imagine, we did not make an offer to that candidate. One or more of the things below would have been a much better answer.

Do Less Work. Not the first time you've heard this answer! Clearly we endorse doing less work. The first strategy we can imagine is that we should simply reduce the number of our API calls. If the service is pay-per-request, then this also has the benefit of saving money. While I do not imagine that you are intentionally calling the service more than you need, it's possible that there are some redundant calls you could discover through a review. If that's the case, just eliminate them, avoid the rate limit, and reduce latency in your application by avoiding network calls. As said, though, this is likely to be limited in benefit since there's only so much redundancy you are likely to find. Unless we find that a significant number of calls are redundant, we almost certainly need one of the other solutions in this section.

Caching. An easy way to reduce the number of calls to a remote system is to remember the answers that we've previously received. Again, we've covered caching on more than one occasion in the past, so we do not need to discuss the concept, just where it is applicable and the limitations. The simplest caching strategy would just be used for requests for data and not updates, but we've learned previously about write-through and write-back caches to handle updates. It's possible to make the caching invisible from the point of view of the calling system with something like Redis—a cache that sits outside of the service and can handle requests/responses.

If we do not control the remote system, it may be more difficult to identify when a particular piece of data has changed and a fresh response is needed rather than the cached one. Domain-specific knowledge is helpful here, like with exchange rates. Exchange rates are a price, to some extent: if I'm going to Frankfurt then I need to buy some Euro with my Canadian Dollars. So to buy 100 Euro, the purchase price for me might be $143. Exchange rates vary throughout the day, but perhaps if you request a rate, you get a quote that is valid for 20 minutes. If that's the case, you can add the item to the cache with an expiry of 20 minutes and you can use this value during the validity period without needing to query it again. Other items are harder to manage, of course.

Group Up. The next strategy to consider is whether we can group up requests into one larger request. So instead of, say, five separate requests to update each employee, use one request that updates five employees. The remote API has to allow this; if there is no mass-update API then we're stuck doing them one at a time. The benefit of this may also be limited by how users use the application; if they usually only edit one employee at a time then it does not help because there is nothing else to group the request with. Waiting for other requests might be okay if it's a relatively short wait, but the latency increase does become noticeable for users eventually.

We aren't limited to only grouping related requests. The remote API may let us combine unrelated requests. If that's the case, then we can keep a buffer of requests. When we have enough requests ready to go, we can just send them out together. A typical REST-like API does not necessarily give you the ability to do this sort of thing. And again, it also requires something else to group a given request with; if there's nothing else going on, then how long are we willing to wait?

Grouping requests also overlaps with caching if you choose a write-back style cache or another strategy that allows multiple modifications before the eventual call to update the remote system.

Grouping requests may also make it hard to handle what happens if there's a problem with one of the elements in it. If we asked to update five employees and one of the requests is invalid, is the whole group rejected or just the one employee update? (Is it a transaction?) It also makes it more likely that such an error occurs, since the logic to build a larger request is almost certainly significantly more complex than the logic to build a small request. Similarly, the logic to unpack and interpret a larger response is more complex and more prone to bugs. Misunderstanding a response can easily cause inconsistent data or repeated requests.

These strategies, as with a lot of programming for performance, involve adding implementation complexity in exchange for improved performance. Don't use them when not needed! Simpler implementations are easier to

maintain.

Patience. If the problem is that too many requests are happening in a short period of time, maybe our best solution is to distribute the requests over more time. Assuming time travel hasn't been invented yet and we aren't travelling at relativistic speeds or anything weird like that, the only way for there to be more time is to simply wait. This ultimately means delaying requests to stay under the rate limit. That sounds like the opposite of what we want when we want to go faster, but if we get rate-limit responses the requests are not getting serviced, so we could say that delayed is better than denied.

Outside of the computer context, when there is a more demand for something than capacity, what do we do? That's right, we queue (line up) for it. Here, this means that requests should be added to a queue and a request gets processed when it gets to the front of the queue. Simply controlling the rate at which requests leave the queue is sufficient to ensure that the rate limit is not exceeded. If all requests go through this queue, then it is also a central place to adjust the rate of outgoing requests if the limit changes.

Unsurprisingly, multiple Rust crates do what we want. Below are some examples from the documentation of the `ratelimit` crate[36] version 0.7.1. This is not especially fancy and we could certainly have considered other ones, but it's sufficient for our purposes. The documentation examples cover some implementation details worth talking about. But onward.

```
use ratelimit::Ratelimiter;
use std::time::Duration;

// Constructs a ratelimiter that generates 1 tokens/s with no burst. This
// can be used to produce a steady rate of requests. The ratelimiter starts
// with no tokens available, which means across application restarts, we
// cannot exceed the configured ratelimit.
let ratelimiter = Ratelimiter::builder(1, Duration::from_secs(1))
    .build()
    .unwrap();

// Another use case might be admission control, where we start with some
// initial budget and replenish it periodically. In this example, our
// ratelimiter allows 1000 tokens/hour. For every hour long sliding window,
// no more than 1000 tokens can be acquired. But all tokens can be used in
// a single burst. Additional calls to `try_wait()` will return an error
// until the next token addition.
//
// This is popular approach with public API ratelimits.
let ratelimiter = Ratelimiter::builder(1000, Duration::from_secs(3600))
    .max_tokens(1000)
    .initial_available(1000)
    .build()
    .unwrap();

// For very high rates, we should avoid using too short of an interval due
// to limits of system clock resolution. Instead, it's better to allow some
// burst and add multiple tokens per interval. The resulting ratelimiter
// here generates 50 million tokens/s and allows no more than 50 tokens to
// be acquired in any 1 microsecond long window.
let ratelimiter = Ratelimiter::builder(50, Duration::from_micros(1))
    .max_tokens(50)
    .build()
    .unwrap();

// constructs a ratelimiter that generates 100 tokens/s with no burst
let ratelimiter = Ratelimiter::builder(1, Duration::from_millis(10))
    .build()
    .unwrap();

for _ in 0..10 {
    // a simple sleep-wait
    if let Err(sleep) = ratelimiter.try_wait() {
           std::thread::sleep(sleep);
           continue;
    }
    // do some ratelimited action here
}
```

[36] https://docs.rs/ratelimit/latest/ratelimit/

Enqueueing a request is not always suitable if the user is sitting at the screen and awaiting a response to their request, because it takes a synchronous flow and makes it asynchronous. Rearchitecting the workflows of a user-interactive application may be a larger undertaking than we're willing to do right now in this topic, but it could be a long-term goal for reducing pressure on systems. And even if we can't move all requests to asynchronous, every request that we do make asynchronous takes the pressure off for the ones that need to be synchronous.

For requests that are not urgent, then another option is to schedule the requests for a not-busy time. Applications usually have usage patterns that have busier and less-busy times. So if you know that your application is not used very much overnight, that's a great time to do things that count against your rate limit.

Conversely, CPUs being able to burst their clock speed when they're busy is the opposite of this in some sense. They try to be extra responsive when things are busy, so that the user gets answers more quickly. But the CPU can only keep up the increased tempo for a limited time, until things get too hot.

Imagine that you have a billing system where the monthly invoicing procedure uses the majority of the rate limit. If that happens during the day, then there's no capacity for adding new customers, updating them, paying invoices, etc. The solution then is to run the invoicing procedure overnight instead. Or maybe you can even convince management to make it so not all users are billed on the first of the month, but that might require some charisma. Which leads us to the next idea…

Roll Persuasion. A final option that you may consider is how to get the other side to raise your limit. Sometimes this just means upgrading to a higher billing tier and you get the higher limit immediately. Sometimes it's something you can simply pay for on top of your existing subscription or agreement. You may also be able to negotiate it with the other side if they're open to that, although you might have to be a sufficiently-important customer to even get in the (Zoom) room for that conversation. Throwing money at the problem can actually work so it's worth considering, but it isn't always a realistic option or is maybe just too expensive. (But don't forget: engineer time is expensive too.)

It Happened Anyway?

Despite our best efforts to reduce it, we might still encounter the occasional rate limit. That's not a disaster, as long as we handle it the right way. The right way is not try harder or try more; if we are being rate limited, then we need to try again later. But how much later?

It's possible the API provides the answer in the rate-limit response where it says you may try again after a specific period of time. Waiting that long is then the correct thing to do. If we don't know, though, it makes sense to try an exponential backoff strategy. This strategy is to wait a little bit and try again, and if the error occurs, next time wait a little longer than last time (though exponential implies a multiplicative factor longer), and repeat this procedure until it succeeds.

Exponential backoff is also applicable to unsuccessful requests even if it's not due to rate limiting. If the resource is not available, repeatedly retrying doesn't help. If it's down for maintenance, it could be quite a while before it's back and calling the endpoint 10 times every second doesn't make it come back faster and just wastes effort. Or, if the resource is overloaded right now, the reaction of requesting it more will make it even more overloaded and makes the problem worse! And the more failures have occurred, the longer the wait, which gives the service a chance to recover. Eventually, though, you may have to conclude that there's no point in further retries. At that point you can block the thread or return an error, but setting a cap on the maximum retry attempts is reasonable.

Having read a little bit of the source code of the Crossbeam (see Appendix C) implementation of backoff, they don't seem to have any jitter in the request. Jitter is an improvement on the algorithm, which prevents all threads or callers from retrying at the exact same time. Let me explain with an example: I once wrote a little program that tried synchronizing its threads via the database and had an exponential backoff if the thread in question did not successfully lock the item it wanted. I got a lot of warnings in the log about failing to lock, until I added a little randomness to the delay. It makes sense; if two threads fail at time X and they will both retry at time $X + 5$ then they will just fight over the same row. If one thread retries at $X + 9$ and another $X + 7$, they won't conflict.

The exponential backoff with jitter strategy is good for a scenario where you have lots of independent clients accessing the same resource. If you have one client accessing the resource lots of times, you might want something else; something resembling TCP congestion control. See [Aeo19] for details.

17 — Mostly Data Parallelism

Data and Task Parallelism

There are two broad categories of paralellism: data parallelism and task parallelism. An analogy to data parallelism is hiring a call center to (incompetently) handle large volumes of support calls, *all in the same way*. Assembly lines are an analogy to task parallelism: each worker does a *different* thing.

More precisely, in data parallelism, multiple threads perform the *same* operation on separate data items. For instance, you have a big array and want to double all of the elements. Assign part of the array to each thread. Each thread does the same thing: double array elements.

In task parallelism, multiple threads perform *different* operations on separate data items. So you might have a thread that renders frames and a thread that compresses frames and combines them into a single movie file.

You're not using those bytes, are you?

So as a first idea we might think of saving some space by considering the range of, for example, an integer. An `i32` is 4 bytes. (In C, `int` is usually 4, though only guaranteed to be at least 2). If we have an integer array of capacity N that uses $N \times 4$ bytes and if we want to do something like increment each element, we iterate over the array and increment it, which is a read of 4 and write of 4. Now, if we could live with limiting our maximum value from 2,147,483,647 (signed, or 4,294,967,295 unsigned) to 32,767 (signed, or 65,535 unsigned), we could reduce in half the amount of space needed for this array and make operations like incrementing take half as much time!

Aside from the obvious tradeoff of limiting the maximum value, the other hidden cost is that of course things that were simple like `array[i] += 1` is more complicated. What do we do now?

Instead of `+=1` we need to calculate the new number to add. The interesting part is about how to represent the upper portion of the number. For just adding 1 it might be simple, and we can manually break out our calculators or draw a bit vector or think in hexadecimal about how to convert a number if it's more difficult. But you wouldn't—you would probably just use bit shift to calculate it. But one must be careful with that as well: the bit shift does sign extension which sometimes you don't want (or does unexpected things), and if we have to bit shift on every iteration of the loop, it's not clear that this is better than two assignment statements...

Maybe you think this example is silly because of Rust's `i8`/C's `short` types. Which you could certainly use to reduce the size of the array. But then modifying each `short` in a different instruction defeats the purpose.

Aha! We can also take it a step farther: if it's a 64-bit processor there's no reason why you couldn't modify 8 bytes in a single instruction. The principle is the same, even if the math is a little more complex.

What we've got here is a poor person's version of Single Instruction Multiple Data (SIMD) (in NZ-speak, using No. 8 wire to implement SIMD), because we have to do our own math in advance and/or do a lot of bit shifting every time we want to use a value... This is a pain. Fortunately, we don't have to...

Data Parallelism with SIMD

The "typical" boring standard uniprocessor is Single Instruction Single Data (SISD) but since the mid-1980s we've had more options than that. We'll talk about single-instruction multiple-data (SIMD) later on in this course, but here's a quick look. Each SIMD instruction operates on an entire vector of data. These instructions originated with supercomputers in the 70s. More recently, GPUs; the x86 SSE instructions; the SPARC VIS instructions; and the Power/PowerPC AltiVec instructions all implement SIMD.

SIMD provides an advantage by using a single control unit to command multiple processing units and therefore the amount of overhead in the instruction stream. This is something that we do quite frequently in the everyday: if I asked someone to erase the board[37], it's more efficient if I say "erase these segments of the board" (and clearly indicate which segments) than if I say "erase this one" and when that's done, then say "erase that one"... and so on. So we can probably get some performance benefit out of this!

There is the downside, though, that because there's only the one control unit, all the processing units are told to do the same thing. That might not be what you want, so SIMD is not something we can use in every situation. There are also diminishing returns: the more processing units you have, the less likely it is that you can use all of that power effectively (because it will be less likely to have enough identical operations) [Ton09].

Compilation. Let's look at an example of SIMD instructions when they are compiled.

By default your compiler will assume a particular target architecture; which one exactly is dependent on what the Rust team decided some time in the past. Choosing a too-new architecture will cause your code to fail on older machines. The choice of architecture can be overridden in your compile-time options with the `target` parameter. Let's look at some SSE code to add two slices and put the result in a third slice:

```
pub fn foo(a: &[f64], b: &[f64], c: &mut [f64]) {
    for ((a, b), c) in a.iter().zip(b).zip(c) {
        *c = *a + *b;
    }
}
```

We can compile with `rustc` defaults and get something like this as core loop contents:

```
movsd   xmm0, qword ptr [rcx]
addsd   xmm0, qword ptr [rdx]
movsd   qword ptr [rax], xmm0
```

This uses the SSE[38] register `xmm0` and SSE2 instructions `movsd` and `addsd`; the `sd` suffix denotes scalar double instructions, applying only to the first 64 bits of the 128-bit `xmm0` register—this is a literal translation of the code. If you additionally specify `-O`, the compiler generates a number of variants, including this middle one:

```
movupd  xmm0, xmmword ptr [rdi + 8*rcx]
movupd  xmm1, xmmword ptr [rdi + 8*rcx + 16]
movupd  xmm2, xmmword ptr [rdx + 8*rcx]
addpd   xmm2, xmm0
movupd  xmm0, xmmword ptr [rdx + 8*rcx + 16]
addpd   xmm0, xmm1
movupd  xmmword ptr [r8 + 8*rcx], xmm2
movupd  xmmword ptr [r8 + 8*rcx + 16], xmm0
```

The *packed* operations (`p`) operate on multiple data elements at a time (what kind of parallelism is this?) The implication is that the loop only needs to loop half as many times. The compiler includes more variants, not shown, to handle cases where there are odd numbers of elements in the slices.

So this is a piece of good news, for once: there's automatic use of the SSE instructions if your compiler knows the target machine architecture supports them. However, we can also explicitly invoke these instructions, or use

[37] Classrooms. How 2019.

[38] You can also compile without SIMD using `-target=i586-unknown-linux-gnu` and see the stack-based x87 instructions.

libraries[39], although we won't do that much. Instead, we'll learn more about how they work and then do some measurement as to whether they really do.

SIMD is different from the other types of parallelization we're looking at, since there aren't multiple threads working at once. It is complementary to using threads, and good for cases where loops operate over vectors of data. These loops could also be parallelized; multicore chips can do both, achieving high throughput. SIMD instructions also work well on small data sets, where thread startup cost is too high, while registers are just there.

In [Lem18], Daniel Lemire argues that vector instructions are, in general, a more efficient way to parallelize code than threads. That is, when applicable, they use less overall CPU resources (cores and power) and run faster.

Rust will generally align primitives to their sizes. On some architectures, this may have performance implications (ARM Cortex), but perhaps not on x86 since you were in elementary school. Under the default representation, Rust promises nothing else about alignment. You can use the `repr(packed(N)` or `repr(align(N)` directives to express constraints on alignment, and you can specify the C representation, which allows you more control over data layout.

Worked Example. So let's say that you actually wanted to try it out. Let's consider a `simdeez` example, which I've put in the repo's live coding subdir under `lectures/live-coding/L16`.

```
use simdeez::*;
use simdeez::scalar::*;
use simdeez::sse2::*;
use simdeez::sse41::*;
use simdeez::avx2::*;

simd_runtime_generate!(
// assumes that the input sizes are evenly divisible by VF32_WIDTH
pub fn add(a:&[f32], b: &[f32]) -> Vec<f32> {
  let len = a.len();
  let mut result: Vec<f32> = Vec::with_capacity(len);
  result.set_len(len);
  for i in (0..len).step_by(S::VF32_WIDTH) {
    let a0 = S::loadu_ps(&a[i]);
    let b0 = S::loadu_ps(&b[i]);
    S::storeu_ps(&mut result[0], S::add_ps(a0, b0));
  }
  result
});

fn main() {
  let a : [f32; 4] = [1.0, 2.0, 3.0, 4.0];
  let b : [f32; 4] = [5.0, 6.0, 7.0, 8.0];

  unsafe {
    println!("{:?}", add_sse2(&a, &b))
  }
}
```

What this does is generate an `add_*` function for each of `scalar`, `sse2`, `sse41`, and `avx`. Then `main` unsafely calls `add_sse2` with two length-4 arrays of `f32`s and gets a `Vec<f32>` back.

`simdeez` is a fairly lightweight wrapper around SIMD instructions and just calls the `loadu_ps` and `storeu_ps` calls to load and store packed single-precision numbers, and `add_ps` to add them. Operator overloading works too.

[39] A discussion of libraries available as of May 2020: `https://www.mdeditor.tw/pl/pdnr`; your choices are `packed_simd` (nightly Rust only), `faster` (unmaintained), or `simdeez` (must use unsafe Rust).

Case Study on SIMD: Stream VByte

"Can you run faster just by trying harder?"

The performance improvements we've seen to date have been leveraging parallelism to improve throughput. Decreasing latency is trickier—it often requires domain-specific tweaks.

Sometimes it's classic computer science: Quantum Flow found a place where they could cache the last element of a list to reduce time complexity for insertion from $O(n^2)$ to $O(n \log n)$.

`https://bugzilla.mozilla.org/show_bug.cgi?id=1350770`

We'll also look at a more involved example of decreasing latency today, Stream VByte [LKR18], and briefly at parts of its C++ implementation. Even this example leverages parallelism—it uses vector instructions. But there are some sequential improvements, e.g. Stream VByte takes care to be predictable for the branch predictor.

Context. We can abstract the problem to that of storing a sequence of small integers. Such sequences are important, for instance, in the context of inverted indexes, which allow fast lookups by term, and support boolean queries which combine terms.

Here is a list of documents and some terms that they contain:

docid	terms
1	dog, cat, cow
2	cat
3	dog, goat
4	cow, cat, goat

The inverted index looks like this:

term	docs
dog	1, 3
cat	1, 2, 4
cow	1, 4
goat	3, 4

Inverted indexes contain many small integers in their lists: it is sufficient to store the delta between a doc id and its successor, and the deltas are typically small if the list of doc ids is sorted. (Going from deltas to original integers takes time logarithmic in the number of integers).

VByte is one of a number of schemes that use a variable number of bytes to store integers. This makes sense when most integers are small, and especially on today's 64-bit processors.

VByte works like this:

- x between 0 and $2^7 - 1$, e.g. $17 = 0b10001$: $0xxxxxxx$, e.g. 00010001;
- x between 2^7 and $2^{14} - 1$, e.g. $1729 = 0b11011000001$: $1xxxxxxx/0xxxxxxx$, e.g. $11000001/00001101$;
- x between 2^{14} and $2^{21} - 1$: $1xxxxxxx/1xxxxxxx/0xxxxxxx$;
- etc.

That is, the control bit, or high-order bit, is 0 if you have finished representing the integer, and 1 if more bits remain. (UTF-8 encodes the length, from 1 to 4, in high-order bits of the first byte.)

It might seem that dealing with variable-byte integers might be harder than dealing fixed-byte integers, and it is. But there are performance benefits: because we are using fewer bits, we can fit more information into our limited

RAM and cache, and even get higher throughput. Storing and reading 0s isn't an effective use of resources. However, a naive algorithm to decode VByte also gives lots of branch mispredictions.

Stream VByte is a variant of VByte which works using SIMD instructions. Science is incremental, and Stream VByte builds on earlier work—masked VByte as well as VARINT-GB and VARINT-G8IU. The innovation in Stream VByte is to store the control and data streams separately.

Stream VByte's control stream uses two bits per integer to represent the size of the integer:

00	1 byte	10	3 bytes
01	2 bytes	11	4 bytes

Each decode iteration reads a byte from the control stream and 16 bytes of data from memory. It uses a lookup table over the possible values of the control stream to decide how many bytes it needs out of the 16 bytes it has read, and then uses SIMD instructions to shuffle the bits each into their own integers. Note that, unlike VByte, Stream VByte uses all 8 bits of each data byte as data.

For instance, if the control stream contains $0b1000\ 1100$, then the data stream contains the following sequence of integer sizes: $3, 1, 4, 1$. Out of the 16 bytes read, this iteration will use 9 bytes; it advances the data pointer by 9. It then uses the SIMD "shuffle" instruction to put the decoded integers from the data stream at known positions in the 128-bit SIMD register; in this case, it pads the first 3-byte integer with 1 byte, then the next 1-byte integer with 3 bytes, etc. Let's say that the input is `0xf823 e127 2524 9748 1b.. .... .... .....`. The 128-bit output is `0x00f8 23e1/0000 0027/2524 9748/0000 001b`, with the /s denoting separation between outputs. The shuffle mask is precomputed and, at execution time, read from an array.

The core of the (C++) implementation uses three SIMD instructions (also available in `simdeez`):

```
uint8_t C = lengthTable[control];
__m128i Data = _mm_loadu_si128 ((__m128i *) databytes);
__m128i Shuf = _mm_loadu_si128(shuffleTable[control]);
Data = _mm_shuffle_epi8(Data, Shuf);
databytes += C; control++;
```

Discussion. The paper [LKR18] includes a number of benchmark results showing how Stream VByte performs better than previous techniques on a realistic input. Let's discuss how it achieves this performance.

- control bytes are sequential: the processor can always prefetch the next control byte, because its location is predictable;
- data bytes are sequential and loaded at high throughput;
- shuffling exploits the instruction set so that it takes 1 cycle;
- control-flow is regular (executing only the tight loop which retrieves/decodes control and data; there are no conditional jumps).

We're exploiting SIMD, so this isn't quite strictly single-threaded performance. Considering branch prediction and caching issues, though, certainly improves single-threaded performance.

SIMD and Planetary Motion

At the moment, I'm not planning to cover this, but you can read more about SIMD in Rust here:

`https://medium.com/@Razican/learning-simd-with-rust-by-finding-planets-b85ccfb724c3`

18 — Compiler Optimizations

Compiler Optimizations

"Is there any such thing as a free lunch?"

Compiler optimizations really do feel like a free lunch. But what do `-O` or `-C opt-level=3` really mean? We'll see some representative compiler optimizations and discuss how they can improve program performance. Because we're talking about Programming for Performance, I'll point out cases that stop compilers from being able to optimize your code. In general, it's better if the compiler automatically does a performance-improving transformation rather than you doing it manually; it's probably a waste of time for you and it also makes your code less readable. Rust lets you force the compiler to do certain optimizations (inlining) even if it might otherwise think it's a bad idea, which is a good compromise when it works.

Enabling compiler optimization. When you want fast binaries, you want to disable debug information and enable compiler optimization. Specify `cargo --release`. You also want link-time optimization (described below) by adding to your `Cargo.toml`:

```
[profile.release]
lto = true
```

About Compiler Optimizations. First of all, "optimization" is a bit of a misnomer, since compilers generally do not generate "optimal" code. They just generate *better* code.

Often, what happens is that the program you literally wrote is too slow. The contract of the compiler (working with the architecture) is to actually execute a program with the same behaviour as yours, but which runs faster. The contract of the compiler does not include any obligations if there is any undefined behaviour.

I looked at `rustc` to confirm that apart from some vectorization, most of Rust's optimization takes place at the backend LLVM level; the `-C opt-level` option mostly sets inline limits and passes the requested optimization level to the backend. Here's what the optimization levels mean:

- 0: no optimizations, also turns on cfg(debug_assertions).
- 1: basic optimizations
- 2: some optimizations
- 3: all optimizations
- "s": optimize for binary size
- "z": optimize for binary size, but also turn off loop vectorization.

Reference material. Since Rust leverages LLVM optimizations, it's good to understand those. Many pages on the Internet describe optimizations. Here's one that contains good examples for C/C++; I've translated appropriate cases to Rust in this lecture.

`http://www.digitalmars.com/ctg/ctgOptimizer.html`

If you happen to be working with C/C++ in the future, you can find a full list of `gcc` options here:

```
http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
```

Scalar Optimizations

General note: we can use `https://godbolt.org/` to investigate what the compiler does. It will be easier to understand if you specify `-C overflow-checks=n`.

By scalar optimizations, I mean optimizations which affect scalar (non-array) operations. Here are some examples.

Constant folding. Probably the simplest optimization one can think of. Tag line: "Why do later something you can do now?" We simply translate:

$$\texttt{i = 1024 * 1024} \quad \Longrightarrow \quad \texttt{i = 1048576}$$

Enabled always. The compiler will not emit code that does the multiplication at runtime. It will simply use the computed value.

Common subexpression elimination. We can do common subexpression elimination when the same expression `x op y` is computed more than once, and neither `x` nor `y` change between the two computations. In the below example, we need to compute `c + d` only once.

```
pub fn add(c:i32, d: i32, y:i32, z:i32) -> (i32, i32, i32) {
    let a = (c + d) * y;
    let b = (c + d) * z;
    let w = 3; let x = f(); let y = x;
    let z = w + y;
    return (a, b, z);
}

pub fn f() -> i32 { return 5; }
```

Enabled at level 1.

Constant propagation. Moves constant values from definition to use. The transformation is valid if there are no redefinitions of the variable between the definition and its use. In the above example, we can propagate the constant value 3 to its use in `z = w + y`, yielding `z = 3 + y`.

Copy propagation. A bit more sophisticated than constant propagation—telescopes copies of variables from their definition to their use. This usually runs after CSE. Using it, we can replace the last statement with `z = w + x`. If we run both constant and copy propagation together, we get `z = 3 + x`.

In C, these scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`. Fortunately, we're not talking about C here. Unfortunately, probably the LLVM backend that does these optimizations does not know about the guarantees provided by uniqueness.

Redundant Code Optimizations. In some sense, most optimizations remove redundant code, but one particular optimization is *dead code elimination*, which removes code that is guaranteed to not execute. For instance:

```
pub fn f(x:i32) -> i32 {
  return x * 2;
}
```

```
pub fn g() {
  if f(5) % 2 == 0 {
    // do stuff...
  } else {
    // do other stuff
  }
}
```

We see that the then-branch in `g()` is always going to execute, and the else-branch is never going to execute. The general problem, as with many other compiler problems, is undecidable. Let's not get too caught up in the

semantics of the *Entscheidungsproblem*, even if you do speak German and like to show it off by pronouncing that word correctly.

Loop Optimizations

Loop optimizations are often a win, because programs spend a lot of time looping. They are particularly profitable for loops with high iteration counts. The trick is to find which loops those are. Profiling is helpful.

A loop induction variable is a variable that varies on each iteration of the loop; a `for` loop variable is definitely a loop induction variable, but there may be others, which may be functions computable from a primary induction variable. *Induction variable elimination* finds and eliminates (of course!) extra induction variables.

Scalar replacement replaces an array read `a[i]` occuring multiple times with a single read `temp = a[i]` and references to `temp` otherwise. It needs to know that `a[i]` won't change between reads.

Sane languages include array bounds checks, and loop optimizations can eliminate array bounds checks if they can prove that the loop never iterates past the array bounds. This doesn't come up in idiomatic Rust because you would usually iterate on an `IntoIterator`. Language design for the win.

Loop unrolling. This optimization lets the processor run more code without having to branch as often. *Software pipelining* is a synergistic optimization, which allows multiple iterations of a loop to proceed in parallel. This optimization is also useful for SIMD. Rust does this. Here's an example.

```
for i in &[1,2,3,4] {
  f(*i);                    ⟹     f(0); f(1); f(2); f(3);
}
```

Loop interchange. This optimization can give big wins for caches (which are key); it changes the nesting of loops to coincide with the ordering of array elements in memory. Although Rust supports 2-dimensional arrays, it looks like it can be idiomatic Rust to index manually (i.e. `[i*N+j]`) or to use a crate to do it for you, e.g. `ndarray`. For instance:

```
pub fn mul(a: &mut [[i32; 8]; 4], c: i32) {
    for i in 0..8 {
        for j in 0..4 {
            a[j][i] = a[j][i] * c;
        }
    }
}
```

⟹

```
pub fn mul(a: &mut [[i32; 8]; 4], c: i32) {
    for j in 0..4 {
        for i in 0..8 {
            a[j][i] = a[j][i] * c;
        }
    }
}
```

Rust is row-major (a[1][1] is beside a[1][2]) as items in a slice are laid out an equal distance from each other: `https://doc.rust-lang.org/std/primitive.slice.html`. OpenGL, on the other hand, is supposedly column-major.

Strangely enough, sometimes you want to do things the column-major way even though it's "wrong". If your two dimensional array is of an appropriate size then by intentionally hitting things in the "wrong" order, you'll trigger all your page faults up front and load all your pages into cache and then you can go wild. This was suggested as a way to make matrix multiplication faster for a sufficiently large matrix: `https://www.intel.com/content/www/us/en/developer/articles/technical/loop-optimizations-where-blocks-are-required.html?wapkw=loop%20optimization...`

Loop fusion. Here, we transform

```
for i in 0..100 {
   a[i] = 4;
}

for i in 0..100 {
   b[i] = 7;
}
```

$\Longrightarrow$

```
for i in 0..100 {
   a[i] = 4;
   b[i] = 7;
}
```

There's a trade-off between data locality and loop overhead; hence, sometimes the inverse transformation, *loop fission*, will improve performance.

Loop-invariant code motion. Also known as *loop hoisting*, this optimization moves calculations out of a loop.

```
for i in 0..100 {
    s = x * y;
    a[i] = s * i;
}
```

$\Longrightarrow$

```
s = x * y;
for i in 0..100 {
    a[i] = s * i;
}
```

This reduces the amount of work we have to do for each iteration of the loop.

Miscellaneous Low-Level Optimizations

Some optimizations affect low level code generation; here are the ones that `rustc` can do.

Cold. I used to talk about likely/unlikely branch prediction hints, but Rust seems not keen to expose this. Rust does expose the `#[cold]` attribute, which you can use to mark a method as unlikely to be called (e.g. `panic`).

Architecture-Specific. LLVM can generate code tuned to particular processors and processor variants (by using instructions available for certain processors, and by modifying the cost model). You can specify this using `-C target-cpu` and `-C target-feature`. This will enable specific instructions that not all CPUs support (e.g. SSE4.2). `native` is a good target CPU if you're running where you compile. See [Wil20] for a more detailed discussion.

Good to use on your local machine or your cloud servers, not ideal for code you ship to others.

Interprocedural Analysis and Link-Time Optimizations

"Are economies of scale real?"

In this context, does a whole-program optimization really improve your program? We'll start by first talking about some information that is critical for whole-program optimizations. They are much less of an issue for Rust but you may well be programming in C or C++ someday soon.

Alias and Pointer Analysis

I made passing references above to the fact that compiler optimizations often need to know about what parts of memory each statement accesses—things like "neither `x` nor `y` change". This is easy to establish when talking about scalar variables which are stored on the stack. This is much harder in conventional languages when talking about pointers or arrays, which can alias. The whole borrowing thing primarily controls aliasing.

Alias analysis helps by declaring that a given variable `p` does not alias another variable `q`; that is, they point to different heap locations. *Pointer analysis* abstractly tracks what regions of the heap each variable points to. A region of the heap may be the memory allocated at a particular program point.

When we know that two pointers don't alias, then we know that their effects are independent, so it's correct to move things around. This also helps in reasoning about side effects and enabling reordering.

Automatic parallelization is a thing. In general, it's hard. Rayon does it a bit (brief mentions to it in Appendix C). In Rust, controlled aliasing makes automatic parallelization much more tractable. Shape analysis builds on pointer analysis to determine that data structures are indeed trees rather than lists.

For a Rust-centric discussion: `https://doc.rust-lang.org/nomicon/aliasing.html`.

Call Graphs. Many interprocedural analyses require accurate call graphs. A call graph is a directed graph showing relationships between functions. It's easy to compute a call graph when you have C-style function calls. It's much harder when you have virtual methods, as in C++ or Java, or even C function pointers. In particular, you need pointer analysis information to construct the call graph. For Rust, indirect function calls (function pointers) and dynamic dispatch through traits are challenges to call graph construction[40].

Devirtualization. This optimization attempts to convert virtual function calls to direct calls. Virtual method calls have the potential to be slow, because there is effectively a branch to predict. If the branch prediction goes well, then it doesn't impose more runtime cost. However, the branch prediction might go poorly. (In general for both Rust and C++, the program must read the object's vtable.) Plus, virtual calls impede other optimizations. Compilers can help by doing sophisticated analyses to compute the call graph and by replacing virtual method calls with nonvirtual method calls. Consider the following code[41]:

```
fn flag() -> bool { true }

fn main() {
    let mut to: &dyn Foo = &Bar;
    if flag() { to = &Baz; }
    to.foo();
}

trait Foo { fn foo(&self) -> i32; }

struct Bar;
impl Foo for Bar {
    fn foo(&self) -> i32 { println!("bar"); 0 }
}

struct Baz;
impl Foo for Baz {
    fn foo(&self) -> i32 { println!("baz"); 1 }
}
```

Devirtualization could eliminate vtable access; instead, we could just call `Baz.foo()` directly. By the way, "Rapid Type Analysis" (applied to C++, not sure if it's used in Rust) analyzes the entire program, could hypothetically observe that only `Baz` objects are ever instantiated (not true here), and would in that case enable devirtualization of the `to.foo()` call.

Inlining. We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

In Rust, you can tell the compiler to inline a function using an annotation:

- #[inline] hints the compiler to perform an inline expansion.
- #[inline(always)] asks the compiler to always perform an inline expansion.
- #[inline(never)] asks the compiler to never perform an inline expansion.

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything!

[40] `https://blog.japaric.io/stack-analysis-2/`
[41] Inspired by code in previous footnote.

The Other Side of Inlining. Inlining has one big downside: your program size is going to increase. This is worse than you think: fewer cache hits and therefore more trips to memory. Some inlines can grow very rapidly – just from this your performance may go down.

Note also that inlining is merely a suggestion to compilers [GNU16]. They may ignore you. For C/C++taking the address of an "inline" function and using it; or virtual functions (in C++) will get you ignored quite fast.

Implications of inlining. Inlining can make your life worse in two ways. First, debugging is more difficult (e.g. you can't set a breakpoint in a function that doesn't actually exist). Most compilers simply won't inline code with debugging symbols on. Some do, but typically it's more of a pain.

Second, it can be a problem for library design: if you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (Congratulations, you made a non-binary-compatible change!). This would not be a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Obviously, inlining and devirtualization require call graphs. But so does any analysis that needs to know about the heap effects of functions that get called; for instance, consider this obviously terrible Rust code:

```
static mut N:i32 = 5;

fn f() { }

fn main() {
  unsafe {
    N = 2;
    f();
    println!("{}", N);
  }
}
```

We could propagate the constant value 2 to the print statement, as long as we know that `f()` does not write to `N`. But idiomatic Rust helps us here. If `N` was instead some memory location o with a unique pointer to it, then we would know whether or not `f()` has access to that unique pointer (and, in particular, there wouldn't exist some other pointer also pointing to o). For a shared object, we check whether the callee requests write permission to any object. In any case, we're less likely to have random state hanging around that may or may not be accessed by a function.

Tail Recursion Elimination. This optimization is mandatory in some functional languages; we replace a call by a `goto` at the compiler level. It is not mandatory in C/C++/Rust. Consider this example[42]:

```
pub fn fibonacci(n: u64) -> u64 {
    fn fibonacci_lr(n: u64, a: u64, b: u64) -> u64 {
        match n {
            0 => a,
            _ => fibonacci_lr(n - 1, a + b, a),
        }
    }
    fibonacci_lr(n, 1, 0)
}
```

Here, `fibonacci_lr` doesn't need to return control to its caller (because the recursive call is in tail position, i.e. the last thing that happens in the function). Doing the tail recursion elimination avoids function call overhead and reduces call stack use.

Link-Time Optimizations

Next up: mechanics of interprocedural optimizations in modern open-source compilers. Conceptually, interprocedural optimizations have been well-understood for a while. But practical implementations in open-source compilers are still relatively new; Hubička [Hub14] summarizes more recent history (compared to how long compilers

[42] https://stackoverflow.com/questions/59257543/when-is-tail-recursion-guaranteed-in-rust

have been around). In 2004, the only real interprocedural optimization in gcc was inlining, and it was quite ad-hoc.

The biggest challenge for interprocedural optimizations is scalability, so it fits right in as a topic of discussion for this course. Here's an outline of how it works:

- local generation (parallelizable): compile to Intermediate Representation. Must generate compact IR for whole-program analysis phase.
- whole-program analysis (hard to parallelize!): create call graph, make transformation decisions. Possibly partition the program.
- local transformations (parallelizable): carry out transformations to local IRs, generate object code. Perhaps use call graph partitions to decide optimizations.

There were a number of conceptually-uninteresting implementation challenges to be overcome before gcc could have its intermediate code available for interprocedural analysis (i.e. there was no stable on-disk IR format). The transformations look like this:

- global decisions, local transformations:
  - devirtualization
  - dead variable elimination/dead function elimination
  - field reordering, struct splitting/reorganization
- global decisions, global transformations:
  - cross-module inlining
  - virtual function inlining
  - interprocedural constant propagation

The interesting issues arise from making the whole-program analysis scalable. Firefox, the Linux kernel, and Chromium contain tens of millions of lines of code. Whole-program analysis requires that all of this code (in IR) be available to the analysis and that at least some summary of the code be in memory, along with the call graph. (Since it's a whole-program analysis, any part of the program may affect other parts). The first problem is getting it into memory; loading the IR for tens of millions of lines of code is a non-starter. Clearly, anything that is more expensive than linear time can cause problems. Partitioning the program can help.

How did gcc get better? Hubička [Hub15] explains how. In line with what I've said earlier, it's avoiding unnecessary work.

- gcc 4.5: initial version of LTO;
- gcc 4.6: parallelization; partitioning of the call graph (put closely-related functions together, approximate functions in other partitions); the bottleneck: streaming in types and declarations;
- gcc 4.7–4.9: improve build times, memory usage ["chasing unnecessary data away".]

As far as I can tell, today's gcc, with `-flto`, does work and includes optimizations including constant propagation and function specialization. LLVM and Rust's use of it also include various flavours of LTO. I couldn't find much information about what happens specifically for Rust; I'd expect the LLVM details below to apply. LLVM LTO can, however, optimize across source languages, i.e. if your program contains both C and Rust, the compiler and linker can optimize both using the intermediate representation.

Impact. gcc LTO appears to give 3–5% improvements in performance, which compiler experts consider good. Like we discussed last time, this allows developers to shift their attention from manual factoring of translation units to letting the compiler do it. (This is kind of like going from manual transmissions to automatic transmissions for cars...).

The LLVM project provides more details at [LLV17], while gcc details can be found at [Die09].

19 — Query Optimization

Optimizing Database Queries

Imagine you are given an assignment in a course and you are going to do it now. To get the assignment done, you will probably (1) figure out what exactly the assignment is asking you to do; (2) figure out how you are going to do it (e.g., must do part 1 first because part 2 depends on it...); and finally (3) do it!

This topic is focused on step two of the process: figuring out how to do the assignment. We use the database and query optimization as an example, but the underlying idea of choosing the approach at run-time is applicable to other contexts. This topic will not require you to have taken a databases course, though some familiarity with a SQL database will likely help with understanding. And depending on how the topic was covered in your database course[43], this might be entirely familiar.

The steps for the database server to carry out the query are the same as the steps for how you would do an assignment [SKS11]:

1. Parsing and translation—interpreting the SQL query in a form the computer can work with.
2. Optimization—figuring out how best to carry out the query.
3. Evaluation—execution of the query according to the plan just developed.

The new and interesting part here is that the database server does not just execute a pre-planned series of steps to get the result, but will adapt its approach at run-time based on what it thinks will be most efficient. It is, yes, still executing the executable code of its binary file and that does not change, but the path taken for a given request can and does vary wildly based on factors known only at run-time. How does that happen?

Usually a query is expressed in SQL, and that must then be translated into an equivalent internal expression using relational algebra. Relational algebra, super briefly, is just the set theory representation of database operations. Complex SQL queries are typically turned into *query blocks*, which are translatable into relational algebra expressions. A query block has a single select-from-where expression, as well as related group-by and having clauses; nested queries are a separate query block [EN11].

A query like `SELECT salary FROM employee WHERE salary > 100000;` consists of one query block because it has only one part to it. We have possibilities. We can select all tuples where salary is more than 100 000 and then perform a projection of the salary field of that result (i.e., throw away the fields we do not need). The alternative is to do the projection of salary first and then perform the selection on the cut-down intermediate relation.

Suppose there is a subquery, like `SELECT name, street, city, province, postalCode FROM address WHERE id IN (SELECT addressID FROM employee WHERE department = 'Development');`. Then there are two query blocks, one for the subquery and one for the outer query. If there are multiple query blocks, then the server does not have to follow the same strategy for both.

What we need instead is a *query execution plan*[44]. To build that, each step of the plan needs annotations that specify how to evaluate the operation, including information such as what algorithm or what index to use. An

[43] Or, you referred to my ECE 356 notes on Github to study for it...

[44] `https://www.youtube.com/watch?v=fQk_832EAx4`, or `https://www.youtube.com/watch?v=l3FcbZXn4jM`

algebraic operation with the associated annotations about how to get it done is called an *evaluation primitive*. The sequence of these primitives forms the plan, that is, how exactly to execute the query [SKS11].

If there are multiple possible ways to carry out the plan, which there very often are, then the system will need to make some assessment about which plan is the best. It is not expected that users will write optimal queries; instead the database server should choose the best approach via *query optimization*. Optimization is perhaps the wrong name for this because we are not choosing the *optimal* approach; instead we will make some estimates about the query plans and try to choose the one that is most likely to be best. This suggests, as you may have guessed, we're going to use heuristics and consider trading accuracy for time.

Measures of Query Cost

If you are asked to drive a car from point A to point B and there are multiple routes, you can evaluate your choices. To do so you need to break it down into different sections, such as drive along University Avenue, then get on Highway 85, then merge onto 401... Each segment has a length and a speed, such as knowing that you will drive 4 km along University Avenue and it is signed at 50 km/h (although with red lights and traffic and whatnot the actual average speed may be more like 30 km/h, or even slower than bicycle speed if you time it right). By combining all of the segments, you get an estimate of how long that particular route will take. If you do this for all routes, you can see which route is the best.

Of course, it may turn out that real life gets in the way: if there is a crash on the highway, traffic really sucks and your conclusion that taking this particular route would be fastest turns out to be wrong. Short of being able to see into the future, this is more or less inevitable: estimates are just informed opinions, and things may be worse (or better) than expected.

Where does the time go in executing a query? The biggest component is most likely loading blocks from disk, considering how slow the disk operations are. In reality, CPU time is a nonzero part of query optimization, but we will ignore this (as does [SKS11]) for simplicity's sake and use only the disk accesses to assess cost. The number of block transfers (data moved in and out of memory) and the number of disk seeks (repositioning where on the disk we are reading from) are the important measures of interest here.

We will follow the estimating strategy in [SKS11]. We will imagine the worst case scenario, that is, only one block per table can be in memory at a time. If we are "wrong" and more data can be in memory, the actual cost is less than the estimated cost (which is better than the reverse).

The estimates calculate only the amount of work that we think it will take to complete the operation. Unfortunately, there are several factors that will potentially affect the actual wall-clock time it takes to carry out the plan:

- How busy the system is—if there are multiple concurrent operations then any particular operation may be queued or blocked or otherwise not able to proceed immediately, leading to a longer time to completion.
- What is in the buffer—if partial data is in the buffer that will speed up completion of the operation since some planned disk operations can be skipped.
- Data layout—if the data is packed well on disk then we need to do fewer seek operations (or shorter ones, perhaps); likewise, if the data is distributed over multiple physical disks we can sometimes do some reads in parallel, but it's hard to know exactly how much parallelization is possible.

You can probably think of various other factors. Remember, the estimate is just used to plan how to do the work; it's not a promise of exactly how long it will take.

Note also that the lowest cost approach is not necessarily the fastest. Sometimes we can go faster by using more resources, but the approach the database often takes is the lowest cost (specifically, fewest disk reads). Recalling the earlier driving analogy, you can think of this as perhaps driving a longer route that involves more highway driving and therefore less time, even if it means more fuel consumption due to the increased distance and speed. When driving, we generally prefer to choose the lowest time estimate, but there are also people ("hypermilers") who are really obsessed with getting maximum fuel economy... and the database is one of those people!

Alternative Routes

There exist many different rules that allow transformation of a query into an equivalent. We're not focusing, here, on learning the rules, as that would take us too far into the details about how the database actually works (or be redundant if you took a databases course that covered this already). But equivalency rules exist and resemble expression transformations that we learned in math class. Some of the equivalents are simple, analogous to, for example, multiplication commutes (e.g., $3 \times 10 \times 7$ is the same thing as $7 \times 10 \times 3$) and others seemingly add extra complexity, but might be a useful way to approach the problem (e.g., 14×13 is equivalent to $14 \times 10 + 14 \times 3$).

Suppose our query involves a selection and a join: we want to select the employee number, salary, and address for an employee with an ID of 385. Suppose number and salary are in the employee table with 300 entries, and the address information is in another table with 12000 entries. We have a join query, and if we do this badly, we will compute the join of employees and addresses, producing some 300 results, and then we need to do a selection and a projection on that intermediate relation. If done efficiently, we will do the selection and projection first, meaning the join needs to match exactly one tuple of employees rather than all 300.

The query optimizer should systematically generate equivalent expressions, but since performing all possible transformations and then evaluating each option may itself take non-trivial time, it is likely that the optimizer does not consider every possibility and will take some "shortcuts" rather than brute force this. One technique that helps on top of that is to re-use common subexpressions to reduce the amount of space used by representing the expressions during evaluation [SKS11]. That's an application of the strategy of remembering already-computed results.

It was oversimplifying to have said that choosing a plan was just as simple as picking the one with the lowest cost. There is a little bit more to it than that. The point about choosing the one with the lowest cost is correct (generally), but the difficulty is in devising and calculating all possible evaluation plans. Neither devising nor analyzing alternatives is free in terms of CPU usage or time, and it is possible to waste more time on analysis than a better plan would save.

A simplified approach, then, focuses just on what order in which join operations are done and then how those joins are carried out. The theory is that the join operations are likely to be the slowest and take the longest, so any optimization here is going to have the most potential benefit.

We already know that the order of joins in a statement like $r_1 \bowtie r_2 \bowtie r_3$ (the bowtie symbol means join) is something the optimizer can choose. In this case there are 3 relations and there are 12 different join orderings. In fact, for n relations there are $\frac{(2(n-1))!}{(n-1)!}$ possible orderings [SKS11]. Some of them are symmetric, which reduces the number that we have to calculate, since $r_1 \bowtie r_2$ is not different from $r_2 \bowtie r_1$ (in relational algebra). In any case, even if we can cut down the symmetrical cases, the problem grows out of hand very quickly as n gets larger.

Once more than three relations are affected by a join query it may be an opportunity to stop and think very hard about what is going on here, because this is quite unusual if the database design is good. The database server may want to ask why do you have a join query that goes across six or eight or twelve relations, but the database server (sadly) does not get to write the developers a nasty resignation letter saying that it can't continue to work this hard due to the negative effects on its health. It will dutifully do the work you asked it to and even try to make the best of this inefficient situation by optimizing it. But clearly it cannot examine all (non-symmetric) approaches and choose the optimal one. It would take too long.

Fortunately, we can create an algorithm that can "remember" subsets of the choices. If we have, for example, $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5$ and the database server does not segmentation fault in disgust, we can break that down a bit. We could compute the best order for a subpart, say $(r_1 \bowtie r_2 \bowtie r_3)$ and then re-use that repeatedly for any further joins with r_4 and r_5 [SKS11]. This "saved" result can be re-used repeatedly turning our problem from five relations into two three-relation problems.

This is a really big improvement, actually, considering how quickly the factorial term scales up. The trade-off for this approach is that the resultant approach may not be globally optimal (but instead just locally optimal). If $r_1 \bowtie r_4$ produces very few tuples, it may be maximally efficient to do that join computation first, a strategy that will never be tried in an algorithm where r_1, r_2, and r_3 are combined to a subexpression for evaluation.

Remember, though, this is an estimation process. The previous statement that said $r_1 \bowtie r_4$ produces very few tuples as if it is a fact. The optimizer does not know that for sure and must rely on estimates where available. So

even though the optimizer may, if it had tried all possibilities, have determined that $r_1 \bowtie r_4$ produces the fewest tuples and should be joined first, it is possible that estimate was off and the actual cost of a different plan was lower.

The sort order in which tuples are generated is important if the result will be used in another join. A sort order is called *interesting* if it is useful in a later operation. If r_1 and r_2 are being computed for a join with r_3 it is advantageous if the combined result $r_1 \bowtie r_2$ is sorted on attributes that match to r_3 to make that join more efficient; if it is sorted by some attribute not in r_3 that means an additional sort will be necessary [SKS11].

With this in mind, it means that the best plan for computing a particular subset of the join query is not necessarily the best plan overall, because that extra sort may cost more than was saved by doing the join itself faster. This increases the complexity, obviously, of deciding what is optimal. Fortunately there are, usually anyway, not too many interesting sort orders [SKS11].

Estimating Statistics

For all the discussion about how it might make sense to swap this or change that or do this rather than that, we have not yet really talked about how the system may guess about how many results are going to be returned. In the previous example I used exact numbers, 300... 1... 12000... etc., but for the database server to get those it can either look them up, or it can guess about them. As mentioned earlier, sometimes certain numbers, like the number of rows in a table, are easily available by looking at metadata. If we want to know, however, how many employees have a salary between $40 000 and $50 000, the only way to be sure[45] is to actually do the query, and we most certainly do not want to do the query when estimating the cost, since at that point we might as well not bother optimizing at all.

If we cannot measure, then, well, we need to guess. Estimates are based on assumptions, and those assumptions are very often wrong. That is okay. We do not need to be perfect. All we need is to be better than *not* optimizing. And even if we pick the second or third or fifth best option, that is acceptable as long as we are close to the best option, more often than not.

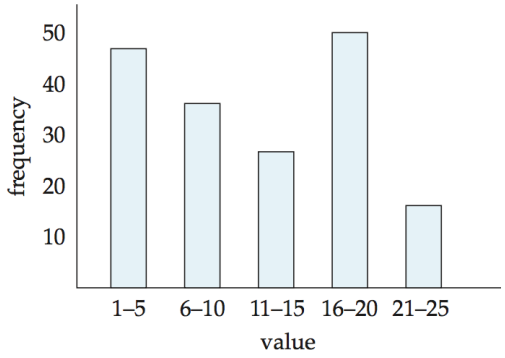
Metadata. As previously mentioned, there is some metadata stored in the database that we could look at to find out some data that we know has some high accuracy. Some items that might be in the metadata, from [SKS11] and [EN11]:

- n_r: the number of rows in a table r
- b_r: The number of blocks containing a table r
- l_r: the size in bytes of table r
- f_r: the number of rows of r that fit into one block
- $V(A, r)$: the number of distinct values in r of attribute A
- $h_{r,i}$: the height of an index i defined on table r

Some of these values can be computed, notably l_r is the number of blocks times the size of a block, and f_r is the number of rows divided by the number of blocks. The value of $V(A, r)$ may or may not be maintained for all attributes, or for groups if that is so desired. If it is on a key field, every value is unique, so we know it is the same as the number of rows for that field. There can also be metadata about index information as well... which might make it metametadata?

A database may also be interested in keeping some statistical information in a histogram. The values are divided into ranges and we have some idea of how many tuples are in those ranges. You have almost certainly seen a histogram before in something like a population pyramid diagram. An example from [SKS11]:

[45] Other than nuking it from orbit...



That should also tell you that they do not necessarily have an even distribution. A histogram does not take up a lot of space and it can help us to figure out certain problems: if we have a histogram of salaries and the largest bucket is 100 000+ and there are 10 people in this category, and there is a query asking how many employees have a salary greater than or equal to $100 000 we will know at least that the number of tuples to be returned is 10. Nice.

The above numbers are exact values which we can know and, hopefully, trust although they could be slightly out of date depending on when exactly metadata updates are performed. The more exact values we have, the better our guesses. But things start to get interesting when, in the previous example, we ask something that does not have a category, such as how many people have a salary larger than $150 000, where there isn't an obvious answer found in the metadata?

Join Elimination and Making a Nested Subquery: I know a shortcut

Join elimination is simply the idea of replacing a query that has a join (expected to be expensive) with an equivalent that does not (expected to be better). It can also turn a join query into a one with a nested subquery, on the theory that two smaller queries might be easier to carry out than a big join. This is a small extension of the idea of choosing the best route to complete the request, because it's more like rewriting the original request to be a little different.

You may ask, of course, why should the optimizer do this work at all? Why not simply count on the developers who wrote the SQL in the first place to refactor/change it so that it is no longer so inefficient? Grind leetcode[46] and use a better algorithm.

SQL is a language in which you specify the result that you want, not the steps for how to get it. If there is a more efficient route, then it's worth taking from the point of view of the database server. The same logic applies in the compiler; if you ask for some operation that the compiler knows it can replace with an equivalent but faster operation, why wouldn't you want that? Compilers don't admonish the user for writing code that it has to transform into a faster equivalent, they just do that transparently. You're welcome!

Obviously, the more complex the query, the harder it is to determine whether or not a particular join may be eliminated. Both foreign key and not null constraints, for example, are beneficial. This reveals a second purpose why constraints are valuable in the database: they enforce logical rules of the application inside of the database, and they allow queries to be completed more efficiently.

Perhaps an analogy helps. You are asked to search through the library to find all copies of the book "Harry Potter and the pthread House Elves". That is a plausible task. But, suppose that you know as well there is a rule that this library will keep only one copy of that book ever. If that is the case, as soon as you have found the single copy of that book, you can stop looking (no need to check more "just in case"). This sort of optimization is very similar in that the rules let us avoid doing unnecessary work and that is a big part of the optimization routine.

[46] For the record, I don't think grind leetcode to get hired is a great plan, and I don't like it when companies expect that of you. It's very artificial. In my experience, most of the time, the challenge lies in understanding the requirements of the work and delivering a good experience (to users in the UI, other developers via API, to your future self/team if you want to build on this, etc...), not writing a provably optimal implementation. I get the impression leetcode interviews are as much hazing as actual assessment of your skills.

Shortcuts

To close out the topic, let's talk about some heuristic rules (guidelines, really) for how the database will make decisions about path to choose. Remember that selection means choosing only the rows that we want and projection is getting rid of the columns that we don't want.

Perform selection early. No surprises here: the sooner we do a selection, the fewer tuples are going to result and the fewer tuples are input to any subsequent operations. Performing the selection is almost always an improvement. Chances are we get a lot of benefit out of selection: it can cut a relation down from a very large number of tuples to relatively few (or even one).

There are exceptions, however. One from [SKS11]: suppose the query is a selection on $r \bowtie s$ where we only want attributes that are in s. If we do the selection first and (1) r is small compared to s and (2) there is an index on the join attributes of s, but not on any of the columns we want, then the selection is not so nice. It would throw away some useful information and force a scan on s; it may be better to do the join using the index and then remove the rows we don't want afterwards.

Perform projection early. Analogous to the idea of doing selection early, performing projection early is good because it tosses away information we do not need and means less input to the next operations. Just like selection, however, it is possible the projection throws away an attribute that will be useful. If the query does not ask for the join attribute in the output (e.g., does it matter what a person's address ID is?) then that join attribute will need to be removed from the output. But if removed too soon, it makes it impossible to do the join.

Set limits. Another strategy for making sure we choose something appropriate within a reasonable amount of time is to set a time limit. Optimization has a certain cost and once this cost is exceeded, the process of trying to find something better stops. But how much time to we decide to allocate? A suggested strategy from [SKS11] says to use some heuristic rules to very quickly guess at how long it will be. If it will be very quick then don't bother doing any further searching, just do it. If it will be moderate in cost, then a moderate optimization budget should be allocated. If it is expensive then a larger optimization budget is warranted.

Plan caching. In any busy system, common queries may be repeated over and over again with slightly different parameters. For example, [SKS11] suggests the following sort of query: a student wishes to query what courses they are enrolled in. If one student does this query with a particular value for student ID number, we can re-use that same evaluation plan in the future when another student does the exact same query with her student ID number instead.

The results will be different and this query may be more expensive on the second run if, for example, the second student is taking 7 courses this term and the first student is taking 5. That is expected, all we really needed was an estimate. With that in mind, if we have actually carried out the exact query we can use the actual execution time as the updated estimate for how long it takes (or perhaps an average of the last n executions of that same query if it has high variability).

Applicability

If we're not planning on implementing or understanding a database, is any of this useful? Yes! The database is just a specific example of an implementation and something that we're familiar with. The real lesson is about how to programmatically generate, evaluate, and choose amongst alternative options for accomplishing a goal.

20 — Self-Optimizing Software

Self-Optimizing Software

Our previous discussion about compiler optimizations (and optimizing the compiler) was focused on things that the compiler can do at compile-time. But what about runtime? The compiler can't do much at runtime, but a sufficiently-smart program can change itself at runtime to be better. Better, in this case, meaning faster. But what about change? We'll start with the simple things, and move on to the more complex and harder to get right. The simple stuff has to do with changing what's in memory. We'll advance to changing the configuration. And, finally, we'll consider changing the binary itself.

Caching. Your first thought about how a program might change itself for speed might be something like caching! Suppose that you keep track of the most popular exchange rates in memory, so that they are available faster than by going to the database. The management of the cache will be at runtime; the contents of the cache will be based on the actual usage. And, it will change over time to adapt to the patterns of usage: if today the exchange rate of CAD-EUR is popular, it will appear in the cache, and without any code changes if the exchange rate of CAD-GBP becomes popular, it goes in the cache and becomes faster to access. So, you *do* technically get different behaviour at runtime, but this is not quite what we wanted to talk about in this topic. It's too easy, and your program should probably be doing it already.

Observe and change. The next idea relates to having your program's configuration change at runtime to adapt to observed behaviour. If there are multiple routes to the same outcome, we might decide at runtime which is the best. This is effective because our initial guess might not be correct, but also because conditions can change at any time. In Java, linked lists versus array lists are a quick example. (You want array lists almost all the time).

Just last class, we talked about query processing: given a certain query (`SELECT id FROM... JOIN... WHERE...`), the database server comes up with an execution plan for how to carry it out. For a simple query, there might be only one answer. A more complex query will have multiple correct answers and the database server will do what it considers best. However, that is based on an estimated cost only. The server could, at least once, try out a different strategy and notice if it is better than the original plan. A lot of queries happen many times (or are extremely similar to already-observed queries), so remembering what worked is helpful.

Building on that, the database server could change how it organizes the data based on what would be most efficient for the most common usage patterns. You can do the same in your program. For an analogy, I can sort the Excel sheet with student grades by either student ID (20xxxxxx) or user ID, based on whatever I use more. So if it is during the term and I'm entering grades, I probably use student ID as the way the file is sorted so I can get efficiently where I need to go. And I could always change that organization if it makes sense, such as after the end of term when people may e-mail me (including their user ID, but not student ID). The idea of re-structuring data storage can also apply to files on disk.

The observe-and-change strategy can also apply when invoking external services (external as in "over the internet"). Suppose there are three different servers where we can send messages and we'll measure and remember how long it took to get a response from that server. The fastest server might normally be the one that is the closest geographically, but that server might be very busy, so it might be faster to communicate with a server that's less busy but farther away. Maybe you send 8 out of every 10 messages to the server that was most recently determined to be the fastest, and one to each of the other two servers. You might discover that your current guess at the fastest server is wrong and it's better to switch your primary.

Genetic algorithms. If you've taken one of the ECE 457x courses, you might have covered genetic algorithms in some detail. This isn't going to be a replacement for that, but will just give you an idea of how the idea can be used. First, a quick three-paragraph explainer on genetic algorithms [Whi98].

A genetic algorithm is inspired by the idea of natural selection. Our program is trying to solve a particular problem. A number of candidate solutions are created (usually, randomly) and they are evaluated for their fitness: how well do they solve the problem? Solutions with a higher fitness have a higher chance of continuing forward into the next group of candidates, called the next generation. At each generation, good solutions from the previous generation are combined (if possible) and/or mutated randomly to see if that makes the solution better. This process repeats until a sufficiently-optimal solution is found, or a fixed number of generations have been evaluated. Thus, solutions with good qualities "reproduce" and move forward in the simulation, and those with bad qualities "die out" and we do not continue down that path. If we do this well, eventually we end up with a solution that's good, or at least good enough.

This works for the kind of problem where we, first of all, have some parameters to configure, and there is a large parameter space. If there is nothing to configure, there's nothing to change or evolve. If the space is "too small", we would search the whole space by brute force. We also need a fitness function that allows us to evaluate how well the problem is being solved. This function cannot be binary (pass/fail) because that doesn't show whether a given solution A is better or worse than solution B; instead we want a continuous or discrete (in the mathematical sense) definition of fitness so we can say solution A with 84.1 is better than B with 81.0.

Of course, a genetic algorithm does not necessarily guarantee the best possible outcome. It is possible that the fitness function tends towards a local maximum rather than the global one, so we get a good solution but not the best. Similarly, the fitness of a solution might be evaluated rarely or take a long time, making the process of finding a good solution slow.

Right, with that in mind, you might ask how genetic algorithms help in making your program faster. The typical use for a genetic algorithm is something like designing an antenna or an airplane wing where making random changes gives some numbers. We can do the same with a generic program, if it has the right properties and what we're trying to do is optimize our configuration parameters.

Let's return to the subject of Google Maps. In our earlier discussion on early-phase termination, we tried to brainstorm ideas about how potential routes are generated and how we know if we have enough or a good-enough route. In the discussion, we will imagine that the decision of when to terminate the search is based on when we have a "good enough" solution. Then there are the various parameters that go into generating a solution, which I'll guess to be something like:

- Number of routes to evaluate
- Heuristic for generating routes to evaluate
- Traffic information reported by other motorists, with decay applied for staleness
- Time of day and month, and whether it is a holiday
- Search radius for alternate routes

It is Google, after all, so they probably consider many more parameters or use a completely different mechanism. It might be difficult to choose what the correct values for these parameters are (especially if they vary by time of day, day of the week, on holidays, etc). Changing them by hand probably does not work; we could let a genetic algorithm choose the values based on experimenting, and trade off the quality of the solution against the time to come up with it. We might consider a solution that only comes up with awful routes to fail, even if it gets them nearly instantly. And we might consider the successful solution one that comes up with a route that is optimal or nearly-optimal in the shortest time.

One reason why genetic algorithms might be a good choice for this kind of problem is that the problem is nonlinear: that is, we cannot treat each parameter as an independent variable and change just one and expect that the change in output is only a result of the change of the one input variable and not also an interaction of one variable with others [Whi98].

Perhaps this takes away some of the mystery of genetic algorithms. Maybe you're thinking this isn't really self-optimizing software, it's just optimizing configuration parameters. Let's go up (well, really, down) a level, then. Now we'll move into changing the binary itself.

Hotspot. The previous discussion of compiler optimizations talked about all the things that compiler can do to make the program more efficient. Some of them are always a clear win. Precomputing something or skipping unnecessary code is always going to be better than the alternative. Other optimizations are not. Let's consider the decision about inlining: sometimes it's good, but sometimes it doesn't help or makes things worse. In those cases, the compiler has to take a decision about whether to do it or not and that's what is in the binary.

In JVM languages like Java, the virtual machine itself can do some optimization because of the just-in-time (JIT) compiler. The original program is turned into bytecode, which is some sort of (high-level) intermediate representation, by the Java compiler, and then there's a second chance at runtime! Oracle's documentation tell us that there's actually two different JIT compilers; one for clients and one for servers. The client one produces its output faster, but the code that it produces will be less efficient. The server one takes more time and more resources to produce slightly better code. Clients pay startup costs more often than servers, so the extra time and resources used for the server JIT compiler would not be well spent.

The major advantage that the JIT compiler has is being able to observe the runtime behaviour of the program and then change its decision. If we see that, for example, inlining would be helpful but the original decision was not to do it, we can change that decision. This is helpful in scenarios like inlining, because we'll have the function call overhead every single time and therefore every call to the function increases the penalty of getting the decision wrong. Being able to change our decision is less helpful when it comes to something like a branch prediction, because the hardware will most likely save us if our prediction isn't very good (see our earlier discussion on this topic).

There are actually a few other things that can be done by the JIT compiler at runtime which are not likely to be doable at compile time. In particular, I want to focus on *escape analysis*, *on-stack replacement*, and *intrinsics*, as outlined in [Str18].

Let's start with escape analysis. The purpose of this is to figure out if there are any side effects visible outside of a particular method. This allows some skipping of heap allocation—we can stack allocate non-escaping objects them instead, saving pressure on the garbage collector. The more interesting thing are possible lock optimizations: lock elision, lock coarsening, and nested locking. Lock elision: if the JIT compiler can determine that a lock serves no purpose—e.g., a method or block is tagged as `synchronized` but it can only ever get called from one thread at a time—there's no need for a lock at all, and therefore no setup or acquisition costs. Lock coarsening: if there are sequential blocks sharing the same lock, the compiler can combine them to reduce the amount of locking and unlocking that needs to happen. This reduces overhead and we'll take it where we can. Nested locks: if the same lock is required repeatedly without releasing (in some recursive code, perhaps), this can also be combined so we don't have as much overhead lost to locking and unlocking.

On-stack replacement is a way that the virtual machine can switch between implementations of a particular method. This is helpful in that if a function is identified as important (sometimes called *hot*) because it runs frequently, then a more optimized version can be swapped in. This does have a slight cost, in that the virtual machine has to pause execution briefly, potentially to recompile (though that could be done on a separate thread), and definitely to swap in the new stack frame (as it may be organized differently), but it will be of some benefit in the long run if this function is truly a frequently-executed piece of code.

If you've done debugging in a JVM language like Java or Kotlin you might have seen something like this in action! While you have the debugger attached, you can make changes, compile it, and the JVM will try to swap in the new code for the old code and continue running. This does not always succeed, but if it does it allows you to try a changed version of the program without having to stop and restart. That's particularly helpful when there's a long workflow to reproduce a particular problem and/or the application in question takes a long time to start up.

Intrinsics are highly-optimized versions of code, precompiled for a specific platform (e.g., x86). If a particular piece of code is truly critical, then using that native implementation might be faster, but it might not always be available, depending on your platform. These were originally developed in C++ rather than Java so it's also done without the safety rails of Java. I guess in that sense you could say that C++ is Java's unsafe mode. (Though Java itself also has an undocumented unsafe mode that people don't talk about). That might be a controversial statement. Now there's a different approach called Graal, which turns the Java bytecode directly into machine

code. Adding an intrinsic is a complicated process.

The hotspot approach is probably closest to what you might have imagined by reading the title of this topic: changing the binary code based on observed runtime behaviour of the program. Rust doesn't have the capability of detecting parameters at runtime and swapping binary code then, because it has chosen not to have a runtime like the JVM. While that does mean that there's less overhead in general, the tradeoff is that any decisions made at compile time will remain so. Unless…

Rewriting the binary. Rewriting the binary in a language like C or Rust can happen, but requires us to do more work ourselves, or depend on a library that implements that functionality for us.

One possibility is to have different versions of a compiled block of code at the ready. That is, we direct the compiler, when the program is being built, to make several variants of the same code with different optimization decisions or other tradeoffs. We'll start with a default, and if we determine based on some observation that we need to change, then overwrite that part of the binary in memory with the new version and we've changed the code path that executes!

There are a couple difficulties that that approach, though. Yes, it increases compile time, but that's usually acceptable as a tradeoff, and is of minor concern. The first big question is how many variants of the code will you compile and have prepared? This is still compile-time work so we're not sure what the real data and behaviour will be like at runtime, so we are still guessing, but we get multiple guesses instead of just one. Then we have to make sure a section of code isn't being executed when we want swap it out, but that's possible to do with some locking mechanism or other software coordination.

That approach is rewriting the binary at run-time, but you might be thinking that if it's all precompiled anyway, can't we decide at runtime with if-statements what function to call to get the same effect? I think so, and thus we haven't really reached the real goal which is doing it truly dynamically. Which is an option.

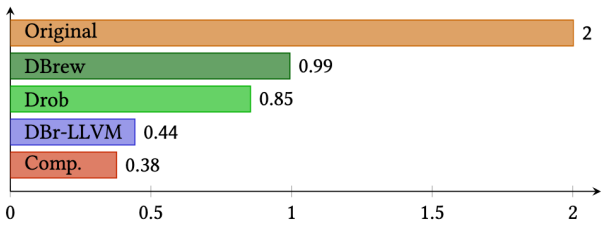
So, you want to compile your own code. With our understanding of all the things the compiler does, you can reasonably assume that you're not going to write your own and include it in your program. Code generation isn't magic or so incredibly difficult that you can't do it, but if you want to make optimal code you need a lot of the analytics and decision-making that's in the compiler. So the only realistic way of getting the code compiled is to use an existing compiler, that is, the one that's already on the target system.

Requiring a specific compiler on the target system might limit your ability to actually deploy this technique. For one thing, if it's end-user software, they might not have compilers installed at all (Windows?), or if they do it's a different compiler (LLVM vs gcc, perhaps)? And you might also encounter a security policy on your servers that forbids the installation of a compiler to make it harder for any exploit that somehow works its way in to compile and run its payload.

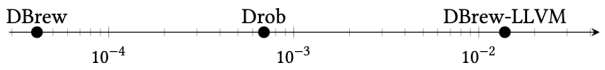
If we can actually use the compiler on the target system, then we are in business. The approach is simple to explain if somewhat complicated in practice. First, take the binary code of the segment that we want to optimize and then have the compiler take a look at it. The compiler will then take the binary, convert it to its internal representation (the intermediate representation), then optimize it, and compile it. The new binary code can get swapped in when it's ready.

On top of the advantage of having runtime information when compiling this time, this kind of approach also makes it possible to inline or rewrite library functions if we want. The amount of benefit will vary a lot based on the library. Some are pretty efficient as-is, but others with lots of functionality might benefit a lot by being optimized for the one use-case you need.

Does this work? Yes; here's some research on the subject from [EHS19]. They are doing a calculation on a matrix of size 649×649 for 20 000 iterations. Here's their graph of the performance results from the improved code:



(a) Execution time [s]



(b) Rewriting and optimization time [s]

As expected, there is a significant cost associated with the process of recompiling a segment of the program. But given the right workload, there is clearly a benefit in the execution time of the updated program. In that case, this is worth doing on code that will run frequently.

Program sus. It's worth noting that programs that rewrite themselves are frequently judged as suspicious by anti-virus and anti-malware software. The story behind this is a part of a software arms race. It starts with the first viruses, which are malicious code doing malicious things. So, anti-virus software is developed to detect viruses and the primary mechanism for detecting if something is a virus is comparing the binary code under examination against a database of malicious software.

To combat this binary-matching pattern, some viruses will alter themselves in subtle ways or produce code that's functionally equivalent but looks different in binary than the original. A virus is not usually something that requires excellent performance, so making a transformation that's slower but looks different to the virus scanner is fine (from the villain point of view, anyway).

It's impossible to predict all possible transformations for a particular piece of software, because there's a lot of routes to the same destination. Anti-virus software can try to ignore things like inserted NOPs or pay no attention to variables X and Y being swapped and then swapped back to just add some assembly instructions. Still, an alternative approach that some anti-virus programs choose is heuristic analysis, which is analyzing broadly what the program under examination does and seeing if its behaviour looks... suspicious.

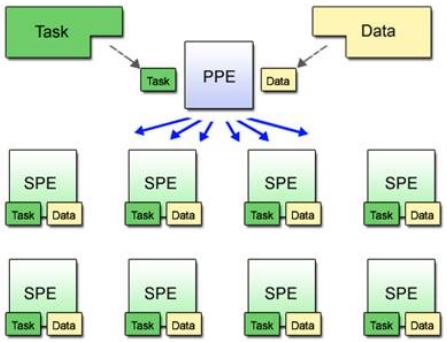
Combining those two facts means that a benign program that changes its own binary code might be considered by anti-virus software to be suspicious and prevented from running or otherwise restricted in some way. If you're determined to rewrite the binary, just keep in mind that anti-malware software end-users have installed may affect the experience (if you are, in fact, shipping to end-users at all).

21 — GPU Programming (CUDA)

GPUs: Heterogeneous Programming

The next part will be about programming for heterogeneous architectures. In particular, we'll talk about GPU programming, as seen in CUDA. The general idea is to leverage vector programming; vendors use the term SIMT (Single Instruction Multiple Thread) to describe this kind of programming. We've talked about the existence of SIMD instructions previously, but now we'll talk about leveraging SIMT more consciously. We are again in the domain of embarrassingly parallel problems.

Cell, CUDA, and OpenCL. Other examples of heterogeneous programming include programming for the PlayStation 3 Cell [Ent08] architecture and CUDA. (Note that the PS4 returns to a regular CPU/GPU configuration; however, it uses AMD hardware which combines the CPU and GPU on one chip.) The Cell includes a PowerPC core as well as 8 SIMD coprocessors:



(from the Linux Cell documentation)

CUDA (Compute Unified Device Architecture) is NVIDIA's architecture for processing on GPUs. "C for CUDA" predates OpenCL; NVIDIA makes CUDA tools available, and they may be faster than OpenCL on NVIDIA hardware. On recent devices, you can use (most) C++ features in CUDA code, which you can't do in OpenCL code. We used to teach OpenCL, but it seems to be the case that CUDA has found widespread acceptance out in industry. Hence, we use CUDA in the course. If you really need cross-platform or you have AMD hardware, then you want OpenCL. The principles are similar enough that you can take what you learned in one toolchain and apply it to the other. Mostly it's syntax.

Programming Model. The programming model for all of these architectures is similar: you write the code for the massively parallel computation (kernel) separately from the main code. Then at run-time, set up the data (input), transfer the data to the GPU, wait while the GPU executes the kernel, then transfer the results back.

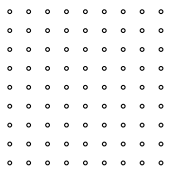
It makes sense to hand it over to the GPU because there are a lot of cores, although they run at a slower speed when compared to the CPU. Looking at the hardware in the ECE servers as of Fall 2020, ecetesla2 has a 4 core 3.6 GHz CPU. It also has 1920 CUDA cores that run at about 1.8 GHz. So, half the speed, but 480 times the workers. Sounds like we could get quite some speedup... if the workload is suitable.

At run-time, there is significant set-up cost for interacting with the GPU and then the data transfers (input to the GPU and collecting the results) mean there is a significant overhead cost for using the GPU. But the GPU can do lots of work in parallel once it gets going. This is a lot like deciding whether to drive or fly.

If the distance is short, say, 200 km (the distance between Ottawa and Montreal) then flying makes no sense: you have to get to the airport, be there at least an hour or two early to make your way through security checkpoints, then fly, then get from the destination airport to your final destination. Sure, the flying part is fast, but the overhead makes your total average speed not worth it.

On the other hand, if you're going a longer distance, like 4000 km (roughly the distance between Waterloo and San Francisco), then driving is way slower! Sure, the overhead of going the airport remains, but once you're in the air you're moving at 800 km/h or so and in 5.5 hours you are there. Compare that to 40 hours of driving.

CUDA includes both task parallelism and data parallelism, as we've discussed earlier in this course. *Data parallelism* is the central feature. You are evaluating a function, or *kernel*, at a set of points. Each point represents a data element, such as one of the bodies in the n-body simulation. But you could also perform some transformation on each pixel of an image or each frame of a video. If you imagine a 2-dimensional problem, each point (little circle) in the diagram below is a so-called *work-item*:



Another name for the set of points is the *index space*.

CUDA also supports *task parallelism*: it can run different kernels in parallel. Such kernels may have a one-point index space. This is doing a large number of different things in parallel. We're not really going to focus on that and will instead stick to the part about data parallelism.

More on work-items. The work-item is the fundamental unit of work. These work-items live on an n-dimensional grid (ND-Range); we've seen a 2-dimensional grid above. You may choose to divide the ND-Range into smaller work-groups, or the system can divide the range for you. CUDA spawns a thread for each work item, with a unique thread ID; they are grouped into blocks. Blocks can share memory locally, but each block has to be able to execute independently. That is, the system can schedule the blocks to run in any order and possibly in parallel (we hope so!).

You get your choice about block size. Usually, we say let the system decide. However, for some computations it's better to specify; if you do, however, you want to make best use of the hardware and use a multiple of the size of the *warp*. A warp is the NVIDIA term for a unit of execution, and there can be multiple units of execution in a given GPU.

Shared memory. CUDA makes lots of different types of memory available to you:

- private memory: available to a single work-item;
- local memory (aka "shared memory"): shared between work-items belonging to the same work-group; created by the compiler and includes values spilled from registers and small arrays declared inside kernels [Mic11]
- global memory: shared between all work-items as well as the host;
- constant memory: resides on the GPU, and cached. Does not change.
- texture memory: this is global memory too, also cached, and it provides potentially a very slight speedup over using global memory. The GPU has texture memory and caches that it uses for rendering and interpolating textures, and it's available for the GPU's general-purpose operations if your use case is a match.

Choosing which kind of memory to use is an important design decision. A simple kernel might put everything in global memory, but that's likely to be slower than making good use of local memory. It could also be tempting to put all unchanging data in the constant memory, but its space is limited, so giant vectors won't always fit.

There is also host memory (RAM in your computer), which generally contains the application's data. We will be doing explicit transfers of the data from host memory to the GPU memory, in both directions.

My very own Kernel. Let's start looking at a kernel. This is the code that will be executed by every CUDA thread somewhat in parallel. First, let's see what the code would look like if we just wrote it in C++:

```
void vector_add(int n, const float *a, const float *b, float *c) {
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}
```

The same code looks like this as a kernel [Cor20]:

```
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = blockIdx.x;
    C[i] = A[i] + B[i];
}
```

We can see in this example that there's no more loop. This is because the loop has become implicit. The loop induction variable here is something we retrieve from `threadIdx.x`—the thread index. The index is a three-component vector so it has x, y, and z dimensions. For this one-dimensional (linear) problem, we only need the first component (x). We'll come back to multi-dimensional problems later.

You can write kernels in a variant of C++, and it looks mostly like the language we (hopefully) know, with a few additions like the `__global__` designation in the previous example. A large number of features of more recent versions of the language are supported, but how much will vary based on your release of the kernel compiler (`nvcc`). There's a list of things that are not allowed in device code. It's unlikely that we'll be doing too many things that are exotic enough to be forbidden by the compiler, in this course. However, if you aren't sure why something isn't working or want to know what kind of stuff is not allowed, see `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrictions`.

The kernel itself is compiled into a form called PTX—Parallel Thread eXecution—instructions. It's possible to write those directly, but we won't; we'll just let the compiler do it. In theory, as with assembly instructions, it might be possible for an expert to produce better PTX instructions than the compiler does, but let's assume we're not. Also, using the `nvcc` compiler makes it so your kernel will run on whatever hardware you have. Just as you would when compiling in another language, it's advised to recompile if you change machines (e.g., uploading to ECE servers from your laptop, or switching from one ECE server to another).

The `nvcc` compiler has specific requirements for the underlying compiler it relies. At the time of writing, the default `gcc` works on `ecetesla[01234]`, so that you can use the command `nvcc -ptx nbody.cu`.

Remember that the kernel has none of the safety guarantees that we normally get in Rust! We can easily read off the end of an array, forget to initialize a variable, allocate memory that we forget to deallocate, or have race conditions, amongst other things. All things that Rust normally prevents. So, not only can you encounter runtime errors in your kernel, but also, it's wise to check that your output matches expectations rather than just assume it's fine. It's even more true here that, just because it compiles, there are no guarantees that it will work.

If you want a quick example of what can go wrong with the kernel at runtime, here's something I got in testing:

```
thread 'main' panicked at 'Failed to deallocate CUDA Device memory.: IllegalAddress',
/home/jzarnett/.cargo/registry/src/github.com-1ecc6299db9ec823/rustacuda-0.1.2/src/memory/device/device_buffer.rs:259:32
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
thread 'main' panicked at 'Failed to deallocate CUDA Device memory.: IllegalAddress',
/home/jzarnett/.cargo/registry/src/github.com-1ecc6299db9ec823/rustacuda-0.1.2/src/memory/device/device_buffer.rs:259:32
stack backtrace:
```

More complex kernel example. Here's a kernel we wrote for an N-body problem. It is just a straightforward brute-force approach (we could approximate to make it go faster, and that was a previous Lab 4):

```
__device__ void body_body_interaction(float4 point1, float4 point2, float3 *acceleration) {
    float4 difference;

    difference.x = point2.x - point1.x;
    difference.y = point2.y - point1.y;
    difference.z = point2.z - point1.z;
    difference.w = 1.0f;

    float distSqr = difference.x * difference.x + difference.y * difference.y + difference.z * difference.z + 1e
        -10;

    float distSixth = distSqr * distSqr * distSqr;
    float invDistCube = 1.0f / sqrtf(distSixth);

    float s = point2.w * invDistCube;

    acceleration->x += difference.x * s;
    acceleration->y += difference.y * s;
    acceleration->z += difference.z * s;
}

extern "C" __global__ void calculate_forces(const float4* positions, float3* accelerations, int num_points) {
    if (blockIdx.x >= num_points) {
        return;
    }
    float4 current = positions[blockIdx.x];
    float3 acc = accelerations[blockIdx.x];

    for (int i = 0; i < num_points; i++) {
        body_body_interaction(current, positions[i], &acc);
    }
    accelerations[blockIdx.x] = acc;
}
```

Let's break it down. The calculation of forces between bodies takes some `float4` and `float3` arguments. In the Rust example, I made my own `Point` and `Acceleration` types. CUDA uses *vector types*, which are a group of n of that primitive type. So a `float4` is a grouping of four `float`s where the components are referred to as `x, y, z, w`. There exist vector types for the standard C primitives (e.g., `int, uint, float, double, char`, and some more) in sizes of 1 through 4. It's just a nice way to package up related values without needing a custom structure (although you can send structures in to kernels). When we get to the host code you'll see that I've had to modify its representation of the data as well.

The function also is prefixed with `__device__` which indicates that it will be called from another function when running on the GPU. On the other hand, `calculate_forces` is a `__global__` function. It is global because the calculation of forces is called from the host; such global functions can call device functions but not other global functions. Device functions can call only other device functions. So it makes it clear where the entry points are from host code. In terms of modularity, you could consider the device functions to be "private" in some sense (but the notion of "module" in this case is sketchy).

The only other thing that really stands out is the `extern "C"` declaration at the beginning of the global function. This disables what is called *name mangling* or *name decoration*, which is to say a compiler trying to differentiate between multiple functions with the same name. If this is too compiler-magic to worry about, just place this magic spell in front of the function call and it prevents the compiler from telling you it can't find the function by the name you specified. More modern versions of `nvcc` may not have this problem, mind you.

We'll also be using the N-body kernel soon as an example for when we examine how to launch a CUDA program.

Also, if you're curious, the compiled version of the kernel resembles assembly language. The output is a little too large to show in the notes/slides, but the compiled version is in the course repository.

Writing a Kernel Well

To start with, of course, we need to decide what sort of work belongs in the kernel to begin with. Anything that would benefit from the very large number of execution units is a good candidate, so that's typically things that are the iterations of a loop. If the loops need to be sequential, though, this is not a good use of the parallelism. If you need a little bit of coordination, we can achieve that, though, with barriers (similar to the concept as seen in concurrency).

Comparing the first kernel that we saw against its CPU-code equivalent, we've taken the explicit for loop and made it implicit as a one-dimensional problem. We just use the GPU on the outer loop. What if we wanted to make it a two-dimensional problem? We could instead treat each pair of points (i, j) as a point in the space, making it a two-dimensional problem. Then you could think of it as a matrix rather than an array and provide it to CUDA like that. This might increase the parallelism!

Although that sounds good, for the N-body problem, it's not. The calculation of body-body interaction for just one pair of points is a very small amount of work, tiny even. Having one work-item for each such calculation means there's a lot of overhead to complete the calculation and it doesn't make sense. But if the calculation of the interaction were more complex, then this transformation might be an improvement.

We can also have three-dimensional problems. If you want something more than a three-dimensional, you have to have some loops in your code. So that 6-level-deep nested-for-loop? You can have the outer three loops as your x, y, and z dimensions, but the rest will be in for loops. Can't avoid everything, I am afraid. The limit to three dimensions is probably because graphics cards were designed for rendering images in 3-D so it seems logical? But that's only a guess.

You can sometimes flatten the dimensions of your problem. A rectangular array in C/C++ is really stored as a linear array and the `[x][y]` is just notational convenience so you could easily just treat it as a linear array. If so, you can avoid the need for a loop in your code. In the N-Body problem, having more items of smaller size is not an improvement because they're too small to be meaningful. Different problems may have different characteristics.

Consider something like brute-forcing a password of 6 characters (easy, but just for the purpose of an example). A one-dimensional approach might generate (in host code) one starting possibility for each valid character in the first position and then let the kernel loop over all possible other values. Then you might be able to improve that by generating starting possibilities in a 3-D matrix for the first three positions and loop in the kernel after that. Whether this is better or not is something that needs testing. And then one could think about generating more starting possibilities by flattening another dimension. It might be faster if it allows more things to be checked in parallel, but it might be slower because of the extra time spent generating the initial conditions in the host code. And eventually at some point (when there are longer passwords), there are too many possibilities to reasonably fit in the host code memory and it becomes impractical to transfer them to and from the GPU.

Of course, this isn't necessarily the best way to brute-force a password. We'll return to that subject later!

Branches. CUDA implements a SIMT architecture. The documentation warns you, however, that unlike a CPU, there's neither branch prediction nor speculative execution. This means that branches are much more costly than they would be on the CPU.

In practice, the hardware will execute all branches that any thread in a warp executed (which can be slow). It then keeps only the results that are valid. Consider this brief example:

```
__global__ void contains_branch(float *a, float *b) {
    if (condition()) {
        a[blockIdx.x] += 5.0;
    } else {
        b[blockIdx.x] += 5.0;
    }
}
```

In the above example, the thread will execute both `if` branches, keeping only the result that is correct. But still, unnecessary work is done. It won't be possible to avoid all conditional branches, but they get expensive quickly.

Similarly, executing a loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item. The compiler will try to unroll loops if they have a known number of iterations. Here, an example from the CUDA docs of a constant-number-of-iterations loop it can unroll:

```
__device__ void foo(int *p1, int *p2) {
  for (int i = 0; i < 12; ++i) {
    p1[i] += p2[i]*2;
  }
}
```

Atomic functions. We also mentioned earlier that there's still the possibility of race conditions. This means that if you want to, say, concurrently add to the same location, you need to use atomic functions. The atomic operations are usable on the standard primitive types like integers, floating point numbers, etc. And there are atomic operations for adding, subtracting, min, increment, decrement, as well as compare-and-swap and bitwise operations. Here's a quick example from the docs on a compare-and-swap [Cor20]:

```
__device__ double atomicAdd(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val + __longlong_as_double(assumed)));

    // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
    } while (assumed != old);

    return __longlong_as_double(old);
}
```

Launch?

So far, all we have covered is the theory and then how to write a kernel. To make use of it, we'll have to look at the host code. That's our next topic.

22 — GPU Programming Continued

GPUs: Heterogeneous Programming

Host Code

We've learned about how a kernel works and a bit about how to write one. The next part is the host code. Now, fortunately, we don't have to write the whole program in C++ or C, even though the kernel has to be written in the CUDA variant. We're going to use the Rustacuda library from `https://github.com/bheisler/RustaCUDA`. That allows us to write code in Rust that interfaces with the GPU, and we can limit the interactions with unsafe code as much as possible.

We'll look at a quick example of launching a very simple kernel from the Rustacuda examples[47]:

```
#[macro_use]
extern crate rustacuda;

use rustacuda::prelude::*;
use std::error::Error;
use std::ffi::CString;

fn main() -> Result<(), Box<dyn Error>> {
    // Set up the context, load the module, and create a stream to run kernels in.
    rustacuda::init(CudaFlags::empty())?;
    let device = Device::get_device(0)?;
    let _ctx = Context::create_and_push(ContextFlags::MAP_HOST | ContextFlags::SCHED_AUTO, device)?;

    let ptx = CString::new(include_str!("../resources/add.ptx"))?;
    let module = Module::load_from_string(&ptx)?;
    let stream = Stream::new(StreamFlags::DEFAULT, None)?;

    // Create buffers for data
    let mut in_x = DeviceBuffer::from_slice(&[1.0f32; 10])?;
    let mut in_y = DeviceBuffer::from_slice(&[2.0f32; 10])?;
    let mut out_1 = DeviceBuffer::from_slice(&[0.0f32; 10])?;

    // This kernel adds each element in `in_x` and `in_y` and writes the result into `out`.
    unsafe {
        // Launch the kernel with one block of one thread, no dynamic shared memory on `stream`.
        let result = launch!(module.sum<<<1, 1, 0, stream>>>(
            in_x.as_device_ptr(),
            in_y.as_device_ptr(),
            out_1.as_device_ptr(),
            out_1.len()
        ));
        result?;
    }

    // Kernel launches are asynchronous, so we wait for the kernels to finish executing.
    stream.synchronize()?;

    // Copy the results back to host memory
    let mut out_host = [0.0f32; 10];
    out_1.copy_to(&mut out_host[0..10])?;
```

[47] `https://github.com/bheisler/RustaCUDA/blob/master/examples/launch.rs`

```
    for x in out_host.iter() {
        assert_eq!(3.0 as u32, *x as u32);
    }

    println!("Launched kernel successfully.");
    Ok(())
}
```

And the kernel it corresponds to is[48]:

```
extern "C" __constant__ int my_constant = 314;

extern "C" __global__ void sum(const float* x, const float* y, float* out, int count) {
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < count; i += blockDim.x * gridDim.x) {
        out[i] = x[i] + y[i];
    }
}
```

Walk-through. Let's look at all of the code in the example and explain the terms. For a more detailed explanation of all the steps, see [Cor20].

One thing that we won't see is any explicit call to initialize the runtime. The CUDA runtime is automatically initialized when there's the first call into it. Thus, the first call might report errors that you wouldn't expect from that call, because they are really a setup problem on your system.

Right, in the example, we need to include the prelude for Rustacuda, just as we've seen previously for Rayon (though we moved Rayon to the appendix of the course notes). The prelude imports a bunch of commonly used types to save on having a lot of imports.

First thing we have to do is initialize the API. This has to happen at the start of the program, so no sense in delaying it! At present, there are no flags defined so the call with `CudaFlags::empty()` is the only valid argument to the initialization function.

Then we get a *device*. The device is your graphics card or any other hardware that does the work. It's obviously possible to have more than one compute device, but we'll just take the first one.

Next, in step 3, we request a *context*. The context is described by the documents as analogous to a process. When we launch something within that context, it executes in there and has its own address space, and when the context is destroyed, all its resources are cleaned up. Each host thread may have one context. The call shown is "create and push" because a host thread has a stack of current contexts. We don't actually need the context for any of the later steps, but we just need to be in possession of one. In this simple example, it stays in scope because everything happens in `main`. If you want more structure to your program, then you do have to ensure it doesn't go out of scope and get dropped, because it has to exist for the other functions to work. When it does get dropped, some cleanup actions take place, of course.

The next step is to create our *module*. A module is technically a package of code and data to run on the device, but in this case it just really means our kernel. What we do here is read the compiled PTX code into a C-String, then create the module from that string. It's also possible to load from a file directly.

Once we have created a module, we then create a *stream*. The stream is where we issue commands such as memory copies, kernel launches, etc. Commands on the same stream execute in order, while commands on different streams execute out of order. Commands that do not specify a stream are issued on the *default stream*. The stream is asynchronous, so once a command has been issued, it returns immediately. Each `DeviceBuffer::from_slice()` is a memory copy issued on the default stream. We use the `DEFAULT` flag to issue the kernel launch on the default stream so that it executes after the memory copies[49].

There's one more step before launching the kernel; in step 5, we create some *data buffers*, which are used for moving data to and from the GPU. Remember, CUDA requires explicit communication, so whatever data want to

[48] `https://github.com/bheisler/RustaCUDA/blob/master/resources/add.cu`

[49] RustaCUDA recommends using the `NON_BLOCKING` flag, but this will result in the kernel launch being issued on a different stream than the memory copies, which is not what we want in this case.

provide as input has to be put into a buffer and then the buffer is transferred to the kernel. Whatever data comes as output will be transferred by the GPU into the output buffers we specify.

After all this setup, we can finally launch the kernel. This has to be done in an `unsafe` block, because the launch macro has to be unsafe (unfortunately)—the GPU interfacing code might do things that don't respect Rust's safety requirements. The good news is that the unsafe block is only the launch, limiting the area of extra scrutiny to something small. When we launch, we specify the kernel that's supposed to run as well as the arguments. Each buffer is converted using `as_device_ptr()` so that the contents of the device buffer are provided. For scalar types like the count, no such conversion is necessary and we can just provide the value. Here, we specify the grid size and block size (1 each). We'll be returning to that subject a bit.

Great! We launched the kernel and sent it over to the GPU. This is an asynchronous process, so we could do more stuff here if we need. There's nothing else to do at the moment, so we'll wait for the items in the queue to complete by calling `stream.synchronize()`. Straightforward!

Finally, in the last step, we copy the items out of the buffer and back into host memory. Here, the example code checks that all the values are correct (3.0) and it is! Alright, we have a simple working example of how to setup, launch, and collect results from a CUDA computation!

N-Body Host Code

Here's the corresponding host code for the N-Body problem where we saw the kernel last time. A lot of it will be the same as the example code, but there are some differences that are noteworthy. Let's get started.

```
#[macro_use]
extern crate rustacuda;
extern crate rustacuda_derive;
extern crate rustacuda_core;

use cuda_sys::vector_types::{float3, float4};
use rustacuda::prelude::*;
use std::error::Error;
use std::ffi::CString;
use rand::Rng;
use rustacuda_core::DeviceCopy;
use std::ops::Deref;

/* A Rustification by Jeff Zarnett of a past ECE 459 N-Body assignment.
Originally from GPU Gems, Chapter 31, modified by Patrick Lam,
then CUDA-fied by Jeff Zarnett using the Rustacuda library example code.
 */

const NUM_POINTS: u32 = 100000;
const SPACE: f32 = 1000.0;

struct CudaFloat4(float4);
unsafe impl DeviceCopy for CudaFloat4 {}
impl Deref for CudaFloat4 {
    type Target = float4;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
struct CudaFloat3(float3);
unsafe impl DeviceCopy for CudaFloat3 {}
impl Deref for CudaFloat3 {
    type Target = float3;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

fn main() -> Result<(), Box<dyn Error>> {
    // Set up the context, load the module, and create a stream to run kernels in.
    rustacuda::init(CudaFlags::empty())?;
    let device = Device::get_device(0)?;
    let _ctx = Context::create_and_push(ContextFlags::MAP_HOST | ContextFlags::SCHED_AUTO, device)?;
```

```
    let initial_positions = initialize_positions();
    println! {"Initial positions:"}
    for pt in initial_positions.iter() {
        println! {"({}, {}, {}) [{}]", pt.x, pt.y, pt.z, pt.w};
    }
    let mut accelerations = initialize_accelerations();

    let ptx = CString::new(include_str!("../resources/nbody.ptx"))?;
    let module = Module::load_from_string(&ptx)?;
    let stream = Stream::new(StreamFlags::DEFAULT, None)?;

    // Create buffers for data
    let mut points = DeviceBuffer::from_slice(initial_positions.as_slice())?;
    let mut accel = DeviceBuffer::from_slice(accelerations.as_slice())?;

    unsafe {
        // Launch the kernel with one block for each point, with 1 thread each, no dynamic shared memory on `
            stream`.
        let result = launch!(module.calculate_forces<<<NUM_POINTS, 1, 0, stream>>>(
            points.as_device_ptr(),
            accel.as_device_ptr(),
            points.len()
        ));
        result?;
    }

    // Kernel launches are asynchronous, so we wait for the kernels to finish executing.
    stream.synchronize()?;

    // Copy the results back to host memory
    accel.copy_to(&mut accelerations)?;

    println! {"Accelerations:"}
    for a in accelerations.iter() {
        println! {"({}, {}, {})", a.x, a.y, a.z};
    }
    Ok(())
}

fn initialize_positions() -> Vec<CudaFloat4> {
    let mut result: Vec<CudaFloat4> = Vec::new();
    let mut rng = rand::thread_rng();

    for _i in 0..NUM_POINTS {
        result.push(CudaFloat4 {
            0: float4 {
                x: rng.gen_range(0.0, SPACE),
                y: rng.gen_range(0.0, SPACE),
                z: rng.gen_range(0.0, SPACE),
                w: rng.gen_range(0.01, 100.0),
            }
        });
    }
    result
}

fn initialize_accelerations() -> Vec<CudaFloat3> {
    let mut result: Vec<CudaFloat3> = Vec::new();
    for _i in 0 .. NUM_POINTS {
        result.push(CudaFloat3 {
            0: float3 {
                x: 0f32,
                y: 0f32,
                z: 0f32,
            }
        });
    }
    result
}
```

We mentioned last time that in the kernel we can use vector types like `float4`. If we want to use those in Rust, we have to import them from a library (here cuda-sys) that isn't the same as the Rustacuda library[50]. This gives us the

[50] So at the time that I wrote this, I actually submitted an issue in the Rustacuda library to bring in support for this. Maybe by the time you

`float4` but there's a requirement of the Rustacuda library that any type that we want to send over to the kernel must have the trait `DeviceCopy`. Implementing the trait is promising that the type you have does not contain any pointers to host memory (so if you had a struct that contained a pointer to a buffer, this is not okay). That's because the pointer will be bogus when it is on the GPU device (they don't share memory). I also added the `Deref` trait which makes it so that elements of the array of this type `CudaFloat4` will be easily converted to the type it contains (`float4`) when we operate on it. And the same for the `float3` type.

The other thing worth noting is that the calculation of forces kernel is invoked with a grid size of `NUM_POINTS` and one thread per block. That is to say, there are `NUM_POINTS` (100 000) chunks of work, and each chunk has one thread. If you get this wrong, the code doesn't work as expected: if you put in 1 and 1 for both of these values, then only the first acceleration will be calculated, because we said there's one chunk of work and it's one thread. But what we actually have asked for is to have `NUM_POINTS` chunks and that will get it done.

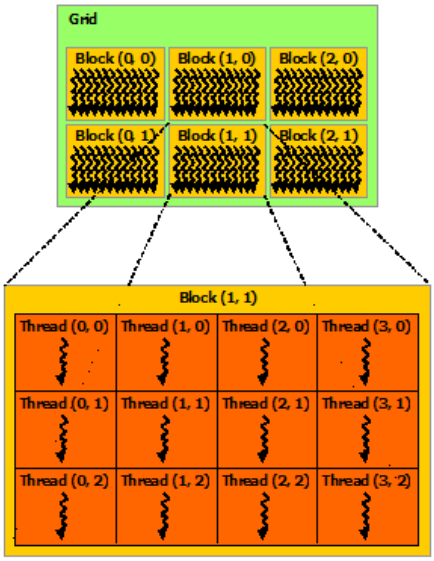
When done, we copy the data out and print it and we're done! Below that, you can also see there's a little change to how the initial values are created in that we have to set the content of the `CudaFloat3/4` through parameter 0.

Putting it to the test. That's great, but how much does it speed up? I ran this on ecetesla1 (because my laptop does not have the correct graphics card for this purpose), so I reran all the CPU benchmarks to have a fair comparison for what happens on the same machine. With 100 000 points:

- `nbody` (sequential, no approximations): 40.3 seconds
- `nbody-parallel` (parallel, no approximations): 5.3 seconds
- `nbody-cuda` (parallel, no approximations): 9.5 seconds

Hey, wait a minute. That's worse than the CPU version. Uh, what's wrong here? My first theory was that 100 000 points is not enough to overcome the overhead of setup and transferring data to and from the device. This turned out to be incorrect (and it was painful to try the sequential version with more points).

Eventually, I used a profiler (we'll come back to profiling shortly) and it told me that most of the time was going to the kernel execution of the `calculate_forces` function. This convinced me that the problem was that I wasn't getting the most out of the GPU as I could be. I don't think there was anything terrible about the kernel itself, but I really wanted to go back to the question of the grids and blocks. Here's a (sadly terrible quality) image from the CUDA docs that explains it [Cor20]:



The documentation isn't super great about how grids and blocks work, unfortunately, and a lot of the guidance on the internet says is "experiment and try" (so helpful). The initial approach that I wrote had each work-item be its own block. That's inefficient, because we're not taking advantage of all the hardware that's available (the warp

are reading this, support has been added. I might even have found time to do it myself, should I finish all the course content and have time.

hardware). The advice that I can find says the number of threads per block should always be a multiple of 32 with a maximum of 512 (or perhaps 1024 with more modern devices). The second guidance I can find is that numbers like 256 and 128 are good ones to start with, and you can tweak it as you need. Then you have to adjust the grid size: divide the number of points by the threads per block to give the number of blocks. Here's the improved call:

```
        let result = launch!(module.calculate_forces<<<(NUM_POINTS/256) + 1, 256, 0, stream>>>(
            points.as_device_ptr(),
            accel.as_device_ptr(),
            points.len()
        ));
        result?;
```

I did have to add a +1 to the number of blocks, because 100 000 does not divide evenly by 256, and if you forget that then the last few of the accelerations are all zeros. But just running it as-is didn't work (and led to the kernel crashing). Why? Because the indexing strategy that I used contained only the reference to the block index `blockIdx.x`. That's fine in the scenario where every work-item gets its own block, but that's no longer the case now: 256 work-items (points) now share each block. Here's the adjusted kernel the calculates the correct index.

```
extern "C" __global__ void calculate_forces(const float4* positions, float3* accelerations, int num_points) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= num_points) {
        return;
    }
    float4 current = positions[idx];
    float3 acc = accelerations[idx];

    for (int i = 0; i < num_points; i++) {
        body_body_interaction(current, positions[i], &acc);
    }
    accelerations[idx] = acc;
}
```

The full version of the improved code is in the course repository as `nbody-cuda-grid`. But what you want to know is, did these changes work? Yes! It sped up the calculation to about 1.65 seconds (still with 100 000 points, still on the same server). Now that's a lot better! We are finally putting the parallel compute power of the GPU to good use and it results in an excellent speedup.

Trading Accuracy for Performance? Thanks to previous ECE 459 student Tony Tascioglu who contributed this section. We've covered on numerous occasions that trading accuracy for performance is often a worthwhile endeavour. You might even say it's a crowd favourite. It's an instructor favourite, at least.

Most of the gaming-oriented NVIDIA GeForce GPUs don't natively support FP64 (double-precision floating point numbers). Native support for that requires expensive datacentre GPUs; it used to be locked in software and is missing in the hardware in more modern cards. Instead of running in hardware, the 64-bit operations are emulated in software and that is significantly slower. How much slower? Using 32-bit floats rather than 64-bit doubles is typically a 16, 32 or even 64× speedup depending on the GPU! We can even push that a bit farther because using a 16-bit float might typically be another 2× faster. For many applications (gaming?) this level of precision isn't necessary.

How dramatic is the difference? See this table from[JeG14], which although its date says 2014, has clearly been updated since then since the GeForce RTX 3080 did not come out until September of 2020:

GPU	FP32 GFLOPS	FP64 GFLOPS	Ratio
GeForce RTX 3090	35580	556	FP64 = 1/64 FP32
GeForce RTX 3080	29770	465	FP64 = 1/64 FP32
Radeon RX 6900 XT	23040	1440	FP64 = 1/16 FP32
Radeon RX 6800 XT	20740	1296	FP64 = 1/16 FP32
GeForce RTX 3070	20310	317	FP64 = 1/64 FP32
GeForce RTX 3060 Ti	16200	253	FP64 = 1/64 FP32

23 — Password Cracking, Bitcoin Mining, LLMs

GPU Application: Password Cracking

GPUs are good—too good, even—at password cracking. We'll discuss a paper that proposes a technique to make it harder to crack passwords. This technique is scrypt, the algorithm behind DogeCoin [Per09]. See also `https://www.tarsnap.com/scrypt.html`

First, let's talk about acceptable practices for password storage. It is *not* acceptable engineering practice to store passwords in plaintext. The inevitable security breach will end with your company sending a "sorry" disclosure email to its clients, and you will be responsible for the ensuing bad publicity. Acceptable practices: **not** plaintext; hashed and salted (we won't discuss salting here but hopefully you remember it from previous courses or other experience.)

Cryptographic hashing. Instead of storing the plaintext password, you store a hash of the password, under a cryptographic hash function. One important property of a cryptographic hash function is that it must be (believed to be a) one-way function; that is: $x \mapsto f(x)$, the forward direction, must be easy to compute, but $f(x) \mapsto x$, the inverse mapping, must be hard to compute. Examples of such functions include SHA-3 and scrypt.

Some known cryptographic algorithms are already pretty well broken (DES, SHA1) and if you choose one of those then it's like no security at all. Other systems have a broken implementation of the algorithm that is vulnerable to some attack. And even if you chose a good algorithm with no known vulnerabilities in the implementation, you need to choose enough bits (e.g., 512 and not 32), otherwise it's too easy to break...

Not Secret. In real life, you can get around the idea of cryptographic hashing by looking on the internet to see if someone's password has already been leaked. Many services are terrible about their password storage policies so if you used the same username and password combination of mycrappywebsite.com and your online banking, then if the mycrappywebsite database gets hacked then the attacker has your username and password already without having to break anything.

First, Check if the Door Is Locked As you might imagine, the first thing to try is super common passwords: "password", "system", et cetera. Users frequently choose common words as passwords and if you just try them all you might get a hit. Choose stronger passwords!

Breaking the hash. Even if there is no known short computation for the inverse function, it's always possible to brute-force the password computation by trying all possible passwords. Think about how GPUs work. Each potential password is a point in the computation space, and we compute the hash over all of them simultaneously. That's a lot of speedup.

Any website with even slightly decent design will start locking accounts after too many bad login attempts, if not outright banning the caller. But if you get a copy of the database, or at least of some cryptographically-hashed passwords, then a brute force approach is possible.

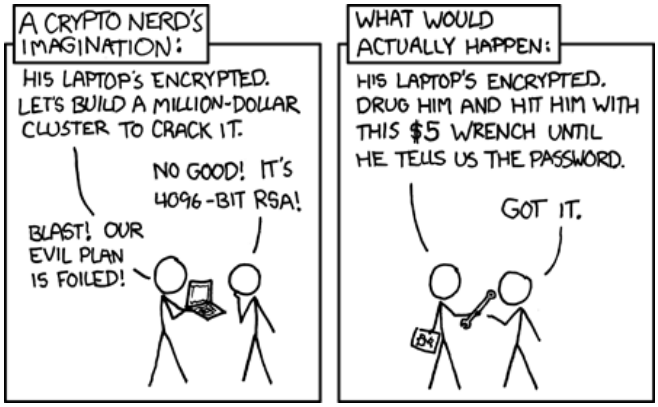
Arms race: making cracking difficult. The idea has always been to make it more difficult to compute the hash function. This does make it longer for the user when they want to log in, but the amount of time to compute a single password is reasonable. However, it's intractable to try all possible passwords, at least with current hardware.

Even way back, UNIX passwords forced repeated applications of the hash function to increase the difficulty. The computational power available to us today is of course dramatically more than it was 20 or 30 years ago, and we can reasonably imagine that the computational power in 20 years will vastly exceed what we currently have, making it plausible to crack a password that's effectively uncrackable today. That's okay, just make sure to change your encryption algorithm (and your password!) as needed to stay ahead of the crackers.

Aside: quantum computing won't basically wreck everything we're talking about in virtually zero time. Those are really good for solving problems like asymmetric key encryption (e.g., RSA), but not as good at hashing problems. Fortunately for our banking details.

The main idea behind scrypt is to make hashing expensive in both time and space, increasing both the number of operations and the cost of brute-forcing. This is how we increase the difficulty to make it implausible to crack in a reasonable amount of time. The only choice that they have to try to break it is, well, to use more circuitry to break passwords (and it will take more time).

Of course, there's always this form of cracking:



(Source: xkcd 538)

Formalization. Let's make the notion of "expensive" a bit more formal. The idea is to force the use of the "most memory possible" for a given number of operations. More memory implies more circuitry required to implement.

Definition 1. *A* memory-hard *algorithm on a Random Access Machine is an algorithm which uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\varepsilon})$.*

Memory-hard algorithms are expensive to implement in either hardware or software.

Now, we want to move from particular algorithms to the underlying functions (that is, we would like to quantify over all possible algorithms). Intuitively, a *sequential memory-hard function* is one where (1) the fastest sequential algorithm is memory-hard; and (2) it is impossible for a parallel algorithm to asymptotically achieve lower cost.

Existence proof. Of course anyone can define anything. It's much better if the thing being defined actually exists. The scrypt paper then goes on to exhibit ReMix, which is a concrete example of a sequential memory hard function.

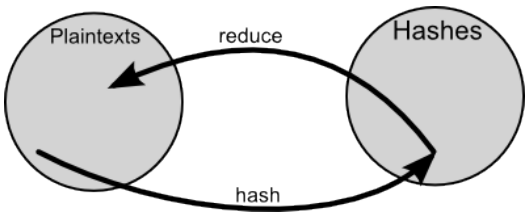
Finally, the paper concludes with an example of a more realistic (cache-aware) model and a hard function in that context, BlockMix.

Rainbow Tables So, the brute force approach is the simplest to describe but is computationally intensive, and if a sufficiently-well-designed cryptographic hash function is used it's really tough to actually crack a password. But maybe if we want to crack a password we don't have to always start from zero; maybe we could remember some

previous computations so that we could use those answers later. If we calculated the hash of password "12345" and we knew what that looked like, then if we encountered that hash in the future we could already jump immediately to the answer in our lookup table. This is the basic idea behind *rainbow tables*.

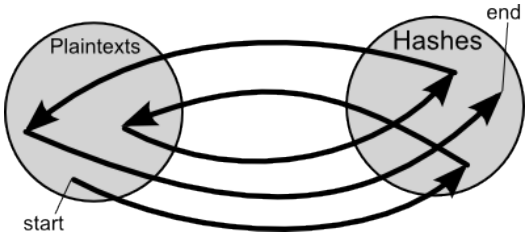
There is a technical paper describing how rainbow tables work, but we'll instead use a much less cryptograhic-expert-level explanation [Kul09].

Part of the difficulty with this approach is that it isn't practical, or even really possible, to store the hashes for every possible plaintext (unless the plaintext is very small). So the rainbow table is a compromise between speed and space. The "reduction" function maps hashes to plaintext:



Showing the reduce and hash functions [Kul09].

This mapping function isn't the inverse of the hash function; it's just some sort of categorization. If the set of passwords to be cracked is, say, six digit numeric, then we compute the hash for a given input ("123456") and we get some output ("d41d8cd98f00b204e9800998ecf8427e") which is then reduced (mapped) to some other value (e.g., we'll take the first 6 numbers, 418980). We have another plaintext now, 418980. So we hash this new one, and reduce it, and so on and so on, until some end point (n times, where you choose n).



And now we have a chain [Kul09].

We should do this to develop some number of chains. This is the sort of task you could do with a GPU, because they can do a reduction relatively efficiently.

Once we have those developed for a specific input set and hash function, they can be re-used forever. You do not even need to make them yourself anymore (if you don't want) because you can download them on the internet... they are not hard to find. They are large, yes, but in the 25–900 GB range, which is large but not ridiculous. I mean, Fallout 76 had a day one patch of 52 GB, and it was a disaster of a game.

Alright, so, you've got them (or made them), but how do we use rainbow tables? Well, for a given hash with an unknown plaintext [Kul09]:

1. Look for the hash in the list of final hashes; if there, break out of the loop
2. If it's not there, reduce the hash into another plaintext and hash the new plaintext
3. Go back to step 1
4. If the hash matches a final hash, the chain with the match contains the original hash
5. Having identified the correct chain, we can start at the beginning of the chain with the starting plaintext and hash, check to see if we are successful (if so, we are done); if not, reduce and try the next plaintext.

Like generation, checking the tables for a hit can also be done efficiently by the GPU. Some numbers from `http://www.cryptohaze.com/gpurainbowcracker.php.html`:

- Table generation on a GTX295 core for MD5 proceeds at around 430M links/sec.
- Cracking a password 'K#n&r4Z': real: 1m51.962s, user: 1m4.740s. sys: 0m15.320s

Yikes. There is obviously a little bit more complexity to how the rainbow tables work (such as dealing with collisions and loops), but it is clear just how devastatingly effective GPU computations are on breaking passwords.

The World is Not Enough

GPUs are great at this, but there are some problems where even the GPU isn't quite the right choice. You have probably guessed it; we're going to talk about Bitcoin. Let's get it out there: I don't think you should mine Bitcoin. This tweet sums up how I would explain Bitcoin to my parents: `https://twitter.com/theophite/status/1030225104234373121?lang=en`... and in any case it's pretty uneconomic to mine it and it's terrible for the environment. At the time of writing, the Bitcoin network's carbon footprint is comparable to that of the entire country of Uzbekistan and it uses electricity comparable to the entire power consumption of the country of Poland (source, and for updated figures, see: `https://digiconomist.net/bitcoin-energy-consumption`). Alright, enough of that: if you want to know more about why you shouldn't mine Bitcoin, talk to me after class.

Our guide in this section is [Tay17], a paper that roughly overviews the history of Bitcoin, how it works, and the trend of mining rigs.

Anyway—Bitcoin is "mined" by doing hash computations, specifically SHA-256. In the beginning, CPUs could be used to mine Bitcoin by performing the calculations. The difficulty of completing the next unit of work increases periodically, so it did not take long for CPU to be inefficient for this purpose. GPUs were the logical step but the quest for more is always ongoing and what do people do when GPU is exhausted?

That's right—they start looking at hardware. Specifically, custom hardware. This works well because the calculations needed are just cryptographic hashing and nothing else. So it's possible to design a system that is optimized to do the few operations in the hash computation (and, xor, rotate, add [modulo], or, right shift) which always happen in a specific order. There's no need for a general purpose CPU or GPU with lots of unnecessary functionality, which just wastes power...

The first hardware miners were built using FPGAs, but they were quickly replaced by ASIC miners. ASIC miners are much more efficient, both in terms of hashes computed per second but also in terms of power consumption. And the more of these that go online, the harder the computation is and the difficulty of mining Bitcoin (in terms of time) increases. These advances make it basically impossible to mine with the hardware you already have.

So now we've uncovered why you shouldn't mine Bitcoin. If you want to do so in a cost-effective manner (otherwise what's the point), you have to spend money on a mining rig of some sort (which is a significant investment), and pay for the power consumption of it (which is also not zero), and some maintenance is required. And because the difficulty is high and new technology is constantly being released to mine more efficiently, it is quite likely that before long your mining setup costs more to run than is earned in Bitcoin. At which point: don't bother.

This isn't a hardware course so we're not going to invest a lot of time in talking about how one might cleverly design hardware. There are other courses for that, but it's the logical extension of using the GPU, so I thought it might be worth a mention.

Large Language Models and You

In November of 2022, OpenAI introduced ChatGPT to the world and suddenly everyone and their best friend was doing some combination of (1) writing news articles about how the advent of AI means we will all be unemployed and lead to the rise of Skynet; (2) founding a startup that used ChatGPT to do something something B2B/B2C

SaaS something something profit; (3) rebranding themselves on LinkedIn as "prompt engineering" experts; or (4) generating lots social media cringe content about how to use it to get rich easily[51].

Such large language models have existed before, but ChatGPT ended up a hit because it's pretty good at being "conversational", which is to say that it does a good job of responding to input in a way that seems like how a human would respond. For this reason, it can be considered to be in the NLP—Natural Language Processing—domain. In the words of some experts [KSK+23]: "These models are trained on massive amounts of text data and are able to generate human-like text, answer questions, and complete other language-related tasks with high accuracy." That is only one of many kinds of large language models and there are many different kinds of machine learning systems out there that do other tasks like recognize images.

Of course, just because such a tool can produce an answer to your question, doesn't mean it is necessarily true or correct. You may enjoy this Legal Eagle video about a lawyer who used ChatGPT for "research" and found that the system returned an answer with completely made-up references. These are sometimes called "hallucinations". We don't have time to watch the video in lecture, but I encourage you to watch it to get an understanding of what went wrong: `https://www.youtube.com/watch?v=oqSYljRYDEM` and why you should not rely on it without checking its output. Remember that in engineering, legally and professionally speaking, the engineer is responsible for understanding how a given tool works and verifying the output is reasonable or correct; a civil engineer who says that the software told them the building was fine will be held liable (both in discipline and legally) if the building was not safe and falls down...

Part of what makes the GPT-3 and GPT-4 models better at producing output that matches our expectations is that it relies on pre-training (it's the PT part of GPT) on a a very large data set, specifically a lot of stuff just out there on the internet [KSK+23]. This course is not one on neural networks, large language models, AI, or similar—there are other such technical electives which you may be taking. This does connect to the course content, however, because generating or updating a pre-trained model is computationally challenging... so performance increases matter here.

Parameters. One factor, but certainly not the only one, in how good the model is at responding to requests is the number of parameters. To explain a bit about why it matters, consider this quote from [MB23]: "LLM AI models are generally compared by the number of parameters — where bigger is usually better. The number of parameters is a measure of the size and the complexity of the model. The more parameters a model has, the more data it can process, learn from, and generate. However, having more parameters also means having more computational and memory resources, and more potential for overfitting or underfitting the data. Parameters are learned or updated during the training process, by using an optimization algorithm that tries to minimize the error or the loss between the predicted and the actual outputs. By adjusting the parameters, the model can improve its performance and accuracy on the given task or domain."

Optimizing LLMs

The content from this section is based on a guide from "Hugging Face" which describes itself as an AI community that wants to democratize the technology. The guide in question is about methods and tools for training using one GPU [Fac23b] (but we can discuss multi-GPU also). Indeed, you may have guessed by the placement of this topic in the course material that the GPU is the right choice for how to generate or train a large language model.

Okay, but why a GPU? In this case we're talking about Transformers and there are three main groups of optimizations that it does [Fac23c]: Tensor Contractions, Statistical Normalizations, and Element-Wise Operators. Contractions involve matrix-matrix multiplications and are the most computationally challenging part of the transform; statistical normalizations are a mapping and reduction operation; and element-wise operators are things like dropout and biases and these are not very computationally-intensive. We don't need to repeat the reasoning as to why GPUs are good at matrix-matrix multiplication and reduction operations since that's already been discussed.

In discussing the optimizations we can make, we'll also need to consider what is in memory, since it's possible that our training of a model might be limited by available GPU memory rather than compute time. Things like the number of parameters and temporary buffers count towards this limit.

[51] Watch this video on the subject of "Get Rich Easy" schemes: `https://www.youtube.com/watch?v=2bq3SdfzcA4`

Optimizing. There are two kinds of optimizations that are worth talking about. The first one is the idea of model performance: how do we generate a model that gives answers or predictions quickly? The second is how can we generate or train the model efficiently.

The first one is easy to motivate and we have learned numerous techniques that could be applied here. Examples: Use more space to reduce CPU usage, optimize for common cases, speculate, et cetera. Some of these are more fun than others: given a particular question, can you guess what the followup might be? Mostly, though, we'll look at how.

Before we get into the subject of how, we should address the question of why you would wish to generate or customize a LLM rather than use an existing one. To start with, you might not want to send your (sensitive) data to a third party for analysis. Still, you can download and use some existing models. So generating a model or refining an existing one may make sense in a situation where you will get better results by creating a more specialized model than the generic one. To illustrate what I mean, ChatGPT will gladly make you a Dungeons & Dragons campaign setting, but you don't need it to have that capability if you want it to analyze your customer behaviours to find the ones who are most likely to be open to upgrading their plan. That extra capability (parameters) takes up space and computational time and a smaller model that gives better answers is more efficient.

What we are going to do is explore the configuration space for training the model. There are a lot of knobs that we can tweak, with respect to which resources to consume. So we'll try to measure the effects of changing resource limits. One challenge, which we'll touch on, is that measurement only works if there is something useful to measure. (Yes, "don't guess, measure", but also you need to measure something meaningful. "Number goes up", in itself, is not useful.)

Our first major optimization, and perhaps the easiest to do, is the batch size. The batch size is just telling the GPU how much to do at once. It's a little bit like when we discussed the idea of creating more threads to increase performance; you may see an improvement by having more workers active but you also may not get any additional benefit from worker $N + 1$ over N since there may not be enough work or other resource conflicts.

I've used an example from Hugging Face [Fac23c] with some light modifications to see what we can do with a very simple example using dummy data. Let's go over and look at that example now. It's in Python (a lot of LLM, machine learning, etc. content is) but it shouldn't be too difficult to understand as we walk through it.

```
import numpy as np
import torch
from datasets import Dataset
from pynvml import *
from transformers import AutoModelForSequenceClassification
from transformers import TrainingArguments, Trainer, logging

default_args = {
    "output_dir": "tmp",
    "evaluation_strategy": "no",
    "num_train_epochs": 1,
    "log_level": "error",
    "report_to": "none",
}


def print_gpu_utilization():
    nvmlInit()
    handle = nvmlDeviceGetHandleByIndex(0)
    info = nvmlDeviceGetMemoryInfo(handle)
    print(f"GPU memory occupied: {info.used // 1024 ** 2} MB.")


def print_summary(res):
    print(f"Time: {res.metrics['train_runtime']:.2f}")
    print(f"Samples/second: {res.metrics['train_samples_per_second']:.2f}")
    print_gpu_utilization()


print("Starting up. Initial GPU utilization:")
print_gpu_utilization()
torch.ones((1, 1)).to("cuda")
print("Initialized Torch; current GPU utilization:")
print_gpu_utilization()
```

```
model = AutoModelForSequenceClassification.from_pretrained("bert-large-uncased").to("cuda")
print_gpu_utilization()

logging.set_verbosity_error()

seq_len, dataset_size = 512, 512
dummy_data = {
    "input_ids": np.random.randint(100, 30000, (dataset_size, seq_len)),
    "labels": np.random.randint(0, 1, dataset_size),
}
ds = Dataset.from_dict(dummy_data)
ds.set_format("pt")

training_args = TrainingArguments(per_device_train_batch_size=4, **default_args)
trainer = Trainer(model=model, args=training_args, train_dataset=ds)
result = trainer.train()
print_summary(result)
```

The bert-large-uncased model [DCLT18] is not a particularly large one – it says on its data sheet that it's about 340 MB – and it's trained on a bunch of English language data. It's uncased because it makes no distinction between capitals and lower-case letters, e.g., it sees "Word" and "word" as equivalent.

First I tried to run it on my laptop, but that failed because it does not have any nvidia GPU, which is not surprising. Next I tried to run this on `ecetesla0` and I saw the following output (skipped some of the stack trace):

```
jzarnett@ecetesla0:~/github/ece459/lectures/live-coding/L24$ python3 dummy_data.py
Starting up. Initial GPU utilization:
GPU memory occupied: 0 MB.
Initialized Torch; current GPU utilization:
GPU memory occupied: 417 MB.
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at
bert-large-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions
and inference.
GPU memory occupied: 1705 MB.
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0;
7.43 GiB total capacity; 6.90 GiB already allocated; 16.81 MiB free; 6.90 GiB
reserved in total by PyTorch) If reserved memory is >> allocated memory try setting
max_split_size_mb to avoid fragmentation.  See documentation for Memory Management
and PYTORCH_CUDA_ALLOC_CONF
```

So the `ecetesla0` machine ran out of memory trying to process this. Using `nvidia-smi` I learned that the card has only `7611MiB` of VRAM available and that does not seem like a lot for the kind of work we are trying to do. The configuration we had asked for a batch size of 4 and it's possible that this is just too much to fit in memory at once for this old card. Reducing batch size to 2 did not help, nor did 1. This is a clear indication that for the model that we want to use, the card isn't going to cut it. Scotty, we need more power.

What I actually did next was change to a smaller version of the model, `bert-base-uncased` which was significantly smaller (110 MB) and something the card could handle. Here's the output with batch size of 1:

```
jzarnett@ecetesla0:~/github/ece459/lectures/live-coding/L24$ python3 dummy_data.py
Starting up. Initial GPU utilization:
GPU memory occupied: 0 MB.
Initialized Torch; current GPU utilization:
GPU memory occupied: 417 MB.
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at
bert-base-uncased and are newly initialized: ['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use
 it for predictions and inference.
GPU memory occupied: 887 MB.
{'loss': 0.0028, 'learning_rate': 1.1718750000000001e-06, 'epoch': 0.98}
{'train_runtime': 109.6152, 'train_samples_per_second': 4.671,
'train_steps_per_second': 4.671, 'train_loss': 0.0027378778694355788, 'epoch': 1.0}
```

```
Time: 109.62
Samples/second: 4.67
GPU memory occupied: 3281 MB.
```

Then I needed to experiment some more with batch size to find the ideal for this card. To condense the results a little bit, see the results table below.

Batch Size	Time (s)	Samples/s	Memory Occupied (MB)	Utilization (%)
1	109.62	4.67	3 281	43.1
2	85.82	5.97	3 391	44.6
4	72.18	7.09	4 613	60.6
8	66.70	7.68	7 069	92.9

And given what we know about the GPU in this system, it's not surprising the OOM error returns when the batch size is increased to 9. The other thing that's nice is that the OOM is encountered very quickly on startup so it's easy to just binary search different batch sizes to find the maximum you can process in one go.

We can try some other optimization techniques to see if we can squeeze out a little more performance from this. There are a number of different techniques that we can focus on to try to optimize memory utilization since that's our limiting factor. Focusing on memory utilization is a part of what makes this topic a little different than most of the others we've covered in this course, which tend to be much more compute-focused.

Gradient Accumulation. The idea behind gradient accumulation is to calculate gradients in small increments rather than for the whole batch; doing this can increase the effective batch size, at the risk of slowing down the total process by having too many compute steps [Fac23b].

Experimenting with this, batch size being fixed at 8:

Gradient Accumulation Steps	Time (s)	Samples/s	Memory Occupied (MB)	Utilization (%)
1	66.06	7.75	7 069	92.9
2	63.96	8.01	7 509	98.7
4	62.81	8.15	7 509	98.7
8	62.65	8.17	7 509	98.7
16	62.42	8.20	7 509	98.7
32	62.44	8.20	7 509	98.7
128	62.20	8.23	6 637	87.2
1024	61.78	8.29	6 637	87.2
4096	62.16	8.24	6 637	87.2

We can see that we very quickly hit diminishing returns on this, but it seems like increasing the number continues to have a marginal benefit, basically for free, up until we get to around 1024. However, I got suspicious about the 128 dropoff in memory usage and it made me think about other indicators—is it getting worse somehow? The output talks about training loss...

Gradient Accumulation Steps	Loss
1	0.029
2	0.070
4	0.163
8	0.169
16	0.447
32	0.445
128	0.435
1024	0.463
4096	0.014

Does that seem concerning? We won't really know unless we do some validation—and this is random data so validating it won't really work for this scenario. Are we perhaps trading accuracy for time? I think the only way to find out is that we need to have a validation data set. We could get through the first steps here of batch size without giving much thought to this part, but now we're kind of stuck. So let's find out.

We'll follow another guide from [Fac23a] where the goal is to train and validate using some Yelp data. Yes, Yelp, the website that struggling restaurant owners blame for ruining their "gourmet burger" place that charges you $22 for an unimpressive reheated Sysco hamburger with no side dish. Running this does take significantly longer, but that's to be expected. The training is divided into three epochs and accuracy is calculated at the end of each of those using a training and evaluation set.

```
import evaluate
import numpy as np
import torch
from datasets import load_dataset
from evaluate import evaluator
from pynvml import *
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import TrainingArguments, Trainer, logging


def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)


def print_gpu_utilization():
    nvmlInit()
    handle = nvmlDeviceGetHandleByIndex(0)
    info = nvmlDeviceGetMemoryInfo(handle)
    print(f"GPU␣memory␣occupied:␣{info.used␣//␣1024␣**␣2}␣MB.")


def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    computed = metric.compute(predictions=predictions, references=labels)
    print(computed)
    return computed


def print_summary(res):
    print(f"Time:␣{res.metrics['train_runtime']:.2f}")
    print(f"Samples/second:␣{res.metrics['train_samples_per_second']:.2f}")
    print_gpu_utilization()


print("Starting␣up.␣Initial␣GPU␣utilization:")
print_gpu_utilization()
torch.ones((1, 1)).to("cuda")
print("Initialized␣Torch;␣current␣GPU␣utilization:")
print_gpu_utilization()

dataset = load_dataset("yelp_review_full")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

tokenized_datasets = dataset.map(tokenize_function, batched=True)

small_train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(1000))
small_eval_dataset = tokenized_datasets["test"].shuffle(seed=42).select(range(1000))

model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=5)
training_args = TrainingArguments(
    per_device_train_batch_size=8,
    gradient_accumulation_steps=1,
    evaluation_strategy="epoch",
    output_dir="test_trainer,"
)
metric = evaluate.load("accuracy")

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
```

```
    eval_dataset=small_eval_dataset,
    compute_metrics=compute_metrics,
)

result = trainer.train()
print_summary(result)
```

And our results table with batch size of 8. I've skipped some intermediate results since at 9 minutes to calculate it takes a while to fill in all the values above. But jumping up some levels illustrates the trend.

Gradient Accumulation Steps	Time (s)	Samples/s	Memory Occupied (MB)	Final Accuracy
1	538.37	5.56	7 069	0.621
8	501.89	5.98	7 509	0.554
32	429.70	6.98	7 509	0.347
1024	513.17	5.85	7 509	0.222

I ran the 32 case a few times to check if this was an outlier—the one in the table is the best result. But it's still noticeably lower than that of the case where gradient accumulation is 1. Interesting, right? Increasing the gradient accumulation does change the effective batch size, and as you may know, increasing the batch size too large means less ability to generalize. Which is another way of saying that the model gets stuck at local minima or overfits the data.

That's not to say that smaller batch sizes are always better; models are way more complicated than that—we can also underfit the model. It's part of why it's important to have training and validation data, so we can optimize and find the right balance. In the Yelp example, I get worse accuracy with batch size of 1 than 4, and 4 is worse than 8. There really is no magic number.

Other Ideas

Just in the interests of time, we won't be able to experiment with everything, but the source has some other ideas that are worth mentioning as they relate to other course concepts we've discussed [Fac23b].

Gradient Checkpointing. This approach is based around the idea of increasing compute time to reduce memory usage. It might allow us to work with a bigger model even with our fairly limited card memory, but training will take longer; according to the source it might be about 20%. By default, all activations from the forward-pass are saved so they can be used in the backward pass; we could not save them and recalculate them from scratch on the backward pass. That would save the most memory but take the most time. A compromise approach is to save some of the activations so that the total amount to recompute on the backward pass is less.

Trying this out with batch size of 8 and gradient accumulation turned off, the total time goes from 66.70 to 93.07s and the memory from 7 069 down to 3619 MB. As expected, we got slower but used less memory. Actually, more like half the memory. Maybe it means we can increase the batch size? Raising it to 16 means the time was 100.55s but still only 3731 MB.

Increasing the batch size a lot to finish faster might work, although it might require a very large batch size and not really save us anything since it would take quite a lot to fall below the time taken when not using the checkpointing. And no, using this checkpointing even with a batch size of 1 is not sufficient to run the `bert-large-uncased` model on `ecetesla0`. And remember that excessively large batch sizes make things worse.

Mixed Precision. This is a fairly straightforward tradeoff of accuracy for time; while the default for most things might be 32-bit floating point numbers, if we don't need that level of precision then some of the 32-bit types could be replaced with 16-bit ones (or smaller!) and this can speed up calculations.

Data Preloading. If your limiting factor is in getting work to the GPU, data pre-loading is about either pinned memory or multi-threads to get data to the GPU faster. If you recall from the operating systems course you (hopefully) took, pinned memory is pages of memory where the operating system is instructed not to swap those pages to disk (i.e., keep them in RAM) for faster access. And multiple threads, well, this is clear at this point.

This is by no means exhaustive—the guide talks about other ideas that we haven't got time to cover, like Mixture of Experts, which are very deep into the details and beyond what we want to cover here. And finally, we could consider doing things like buying a bigger (better) GPU, or using multiple GPUs for more parallelism. All the things we know about CPU work in this problem domain.

Tradeoffs

More than any other topic, the LLM topic shows the inherent tradeoffs in optimizing things. Do we trade memory for CPU? Do we trade accuracy for time? Do we prefer to err on the side of under- or over-fitting the model and how does that affect our choices on the other dimensions? I imagine that in the next few years our tools and ways of deciding these things will become much more sophisticated and best practices and known-good answers will emerge. But in the meantime, we can have a lot of fun experimenting and learning.

24 — Profiling: Observing Operations

Observations Precede Conclusions

Think back to the beginning of the course when we did a quiz on what operations are fast and what operations are not. The important takeaway was not that we needed to focus on how to micro-optimize this abstraction or that hash function, but that our intuition about what is fast and what is slow is often wrong. Not just at a macro level, but at a micro level. You may be able to narrow down that this computation of x is slow, but if you examine it carefully…what parts of it are slow?

If you don't use tools, then you end up guessing. You just make some assumptions about what you think is likely to be slow and try to change it. You've probably heard the famous quotation before, but here it is in its full form:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> – Donald Knuth

So going about this blindly is probably a waste of time. You might be fortunate and optimize a slow part, but we should really follow one of my favourite rules: "don't guess, measure!" So, to make your programs or systems fast, you need to find out what is currently slow and improve it (duh!). Up until now in the course it's mostly been about "let's speed this up", but we did not take much time to decide what we should speed up (though you maybe did this on an assignment…?).

The general idea is, collect some data on what parts of the code are taking up the majority of the time. This can be broken down into looking at what functions get called, or how long functions take, or what's using memory…

Why Observation? We're talking here about the idea of observation of our program, which is a little bit more inclusive than just measuring things, because we may observe things that are hard or impossible to quantify. Observing the behaviour of the program will obviously be super helpful in terms of figuring out what—if anything—to change. We have several different ways of looking at the situation, including logs, counters, profiling, and traces. Differences between them, briefly [Sit21]:

- **Counters** are, well, a stored count of the occurrences of something: how many times we had a cache miss, how many times we called `foo()`, how many times a user logged in...
- **Profiles** are higher-level overviews of what the program is doing, where time is going in aggregate, and how long certain operations take.
- **Traces** are recordings that show the sequence of events that the program is doing, so we can understand the runtime behaviour and how we got to the state we are in.

We'll return to profilers later, because it's a big topic, but for now we'll start with traces and counters.

Tracing: Logging

We'll start with the idea of logging. It's an effective way of finding out what's going on in your program as it executes, and probably all of us have use print statements as a form of debugging, but also as a form of tracing. When we're running a small program, we don't always make the log messages very nice, but in a production system, logs typically have a timestamp, a message, and some attributes. The timestamp is helpful for organizing and searching the logs.

In [Sit21] it's recommended that if there can be only one tool for observing your program's execution, it is logging. This is because logging can tell you things about how much work the software did in a time period, when things are busy, which transactions are slow, when the service is down, etc. This probably matches well with our intuition that printf debugging or tracing is a good first option to figure out what's going on.

Given that we want to log, we do have to consider how often to log, and what should be in the content of it. If we don't log the information, there may not be any good way to recover that information again. But we could also drown out useful information if we log too many (irrelevant) things. Logs typically have levels, like error, warning, info, debug... These can make it easier to spot just the information that is relevant.

A typical approach is to log an incoming request (input) and the outgoing response (output), and maybe some steps in between. We want to link the different things together (perhaps with attributes) so that we can see what happened. Here's a made-up example that doesn't show the time stamps or attributes:

```
Received request: update plan of company 12345 to ULTIMATE
Retrieving company 12345
Verifying eligibility for upgrade of company 12345 from BASIC to ULTIMATE
Company 12345 is not eligible for upgrade due to: unpaid invoices > 0
Returning response: update of company 12345 to ULTIMATE is DECLINED
```

We can see the various steps of the process and quickly understand what happened, because the logs show us the request, the stages of handling the request, and the response. In this example I intentionally added a decision and logged the reason for the decision. That's preferable to making someone guess why the answer was no when looking at the logs.

Choosing the timestamps strategy is also interesting; the level of precision you need in the logging depends on the timescales of your execution. When running processes that take tens of seconds, you could probably be satisfied with down to the thousandth of a second; if things run super fast then we might need microsecond or nanosecond resolution [Sit21]. Time zones also matter, but I'm going to recommend using UTC because it doesn't have weird things like Daylight Savings Time where you can get a jump in the time forward or back. And back is worse because it means a certain minute appears to happen more than once (defeating our normal expectation of linear time). It also avoids issues where the California team thinks it's Monday and the Singapore team thinks it's Tuesday. UTC might be weird, though, if your application is running on a Mars Rover; at that point you might as well use Stardates (Captain's Log, Stardate 46119.1...).

As I already said, logging every request might be too noisy to actually find anything in the resulting data. The ability to search and sort helps, but it can still be going by too fast to realistically assess in real-time. Log aggregation services exist and they can help, especially when trying to do a retrospective.

Typically, adding any relevant attributes is very helpful to identify what has happened or to correlate knowledge. If we can see in the attributes that updates for plans of companies always take ten times longer if the company's address is in Portugal, that's actually a useful observation. We can take that information and put it in the bug ticket and ideally help the developer—even if it's our future selves—to find the issue and resolve it.

With that said: there's such a thing as too much detail in logging. Logs should always redact personally-identifiable information such as people's names, contact information, medical records, etc. Unless the logs are really kept in secure storage with appropriate access controls, they can be seen by people who shouldn't have that information. Why should, for example, a developer on the website team have access to the banking details of a customer? My opinion: they shouldn't. And it's not just my opinion, but also the opinion of people like the Privacy Commissioner of Canada, whom you probably do not want to anger.

One way that we get into the situation of over-logging is if it's difficult or impossible to use other tools on the program or to have (read-)access to the database. If someone is trying to debug the behaviour of the program it might seem sensible to log just about everything because it's frustrating to try to debug a problem with incomplete information.

So we are aiming for a balance: include the relevant information that would let us spot patterns or correlate with other data sources, but not so much that the important information is lost or that PII is exposed.

There is a third way that a program logging too much, beyond drowning out key info or logging PII, and that's if the overhead of tracing is significant. Normally we're happy to accept some small slowdown in our program to have traceability, but let's try to put some numbers to tracing overhead.

Some quick, off-the-cuff figures from [Sit21]:

- If we ask the CPU trace tool to track every conditional branch taken, that would be about a $20\times$ slowdown. This amount of overhead would be acceptable if we're debugging the program on a development machine (own laptop or testing deployment), but certainly not in a production environment.
- If we ask for a timestamp at every function call and return in our program the slowdown is around 1.2–$1.5\times$. This may (emphasis on may) be acceptable in a production environment if the task is not time-critical, but only temporarily while experimenting or observing.
- If we ask for a timestamp of every system call entry and exit (user-space to kernel transition and back), that might be much less than 1% overhead. This would likely be acceptable in a production environment for all but the most time-critical of operations and could remain in place at all times.

That covers tracing the CPU. What about memory? If you've used Valgrind[52] before, you know it. If you haven't, as I say to the ECE 252 students: The target program then runs under the "supervision" of the tool. This results in running dramatically slower than normal, but you get additional checks and monitoring of the program. Why? Valgrind checks every memory read and write for validity (is the variable initialized? Is it off the end of an array? All those fun things that we, fortunately, no longer think about in Rust), but also records information about where memory is allocated to help the programmer in tracking down the genesis of a memory leak. When execution of the program ends, Valgrind shows the stack trace of the path that led to the allocation that was leaked. That's pretty neat!

A disk trace tool would most likely have extremely tiny overhead because the speed of the disk as compared to the CPU is very slow. Something similar goes for the network; the latency is high and speed of the network are slow compared to what the CPU can do [Sit21].

Tracing also can play a role in identifying deadlocks and delays in threads caused by waiting for locks: simply log every lock and unlock. But we might really be only interested in the locks where there is contention, i.e., threads are sometimes or often failing to acquire the lock because it's held by another thread [Sit21]. After all, if our intention is to observe the behaviour of the program with the intention of improving performance, the part where a thread isn't getting what it wants immediately and must wait is much more interesting than the part where everything is fine and there are no delays.

Space, the Final Frontier. Whatever trace strategy we choose, the trace itself takes up space. If we are producing a very large amount of data, it won't all fit in memory and has to go on disk. It's possible that the amount of data that we produce will fill up the disk very quickly, or arrive so fast that the disk cannot keep up. In [Sit21] there's a quick calculation that says a main memory system with 20 GB/s bandwidth and 64-byte cache lines that records up to 8 bytes of data per trace entry could result in producing data at a rate of 2.4 GB per second! That's a rather overwhelming amount of data—even if you could write it all out to disk fast enough, it would fill up an 8 TB hard drive in just under an hour. Either we need to capture less data by recording fewer things or by recording for a shorter time. Just add this to the reasons why we need to be judicious about how much trace data to capture.

[52] So, we used to cover Valgrind in this course. That made sense when it was C and C++ but we used to discuss it in Rust too. While the Helgrind part might help you find lock acquisition issues, for the most part students found that there were too many false positives around memcheck and it was just causing students to stress about losing marks. So... away with it.

Aggregate Measures (Counters)

Many profiling tools rely heavily on counters. As the name suggests, they keep a count of events: interrupts, cache misses, data bytes written, calls to function `do_magic()`. Keeping track of these numbers is relatively inexpensive (specially when compared to other approaches) so counting every occurrence of an event is plausible. Counters are a form of aggregation, because we're summing the number of occurrences and at the end of the program we have a total number. The counter and any data derived from it certainly takes much less space than a trace.

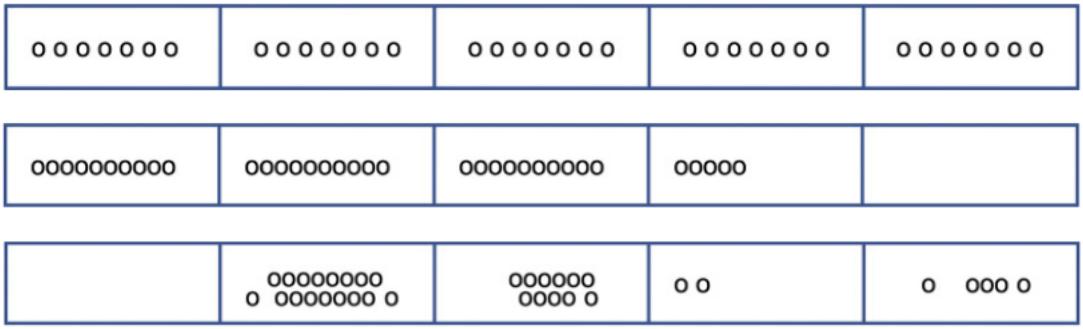
The sum is the simplest kind, but other aggregate measures are exactly what they sound like: summaries of data. Calculating the average response time of a request is an aggregate measure: we've summed up the total time per request and divided it by the number of requests in the given period and hopefully the resulting value is a useful one. Asking the computer to calculate the summary is sensible, of course, because it's not realistic to ask a human to look at 50 000 requests and calculate their average time. Some obvious aggregate measures are things like: number of requests, requests broken down by type, average time to respond to a request, percentage of error responses...

Whatever aggregate measures we use, they are useful only with context. Suppose that after my pull request is merged, the average login time for a user is 0.75 s; is that a problem? Without a baseline to compare against, I'm not sure. If before my PR it was 0.5 s, I made performance much worse and that doesn't sound good; should I revert it and re-work it? Maybe, unless I am intentionally making login slower to make a brute-force password attack more expensive for an attacker. Context is key: the summary tells you some data, but not the reasons.

Another example: if I tell you that a request takes, on average, 1.27 seconds to get a response, is that good or bad? There's, again, no way to say anything about it without a point of reference. Are we being asked to approve or deny a transaction and there's a time limit of 1 second to give our answer (or else a default answer is assumed)? We're missing the target and that's a problem. If instead I said the time limit is 10 seconds, then we have plenty of room. Or do we?

Average time isn't quite enough to know if we're doing well when it comes to time limits. If the average time is 1.27 seconds it does not preclude the possibility that the maximum time is outside the 10 second time limit and that means for some percentage of requests, possibly a large percentage, we're not making the deadline. How hard of a deadline are we talking about? To address this we can look at maximum and 95th percentile kind of aggregate measures as well.

The other way that averages can be misleading is that they misrepresent the bursty nature of the data. If your service receives, on average, 7 requests per second, does that mean it looks like the first, second, or third row below (all from [Sit21])?

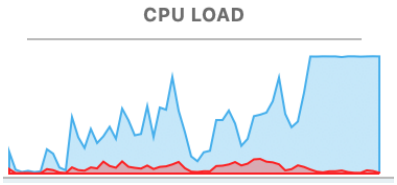


While the first scenario is really nice and very predictable, is that realistic? Later on in the course we'll talk more about queueing theory and we'll learn that random arrivals and what it means for things to arrive.

When we choose to aggregate our data in some way, we must make a decision about the period of time in question. If our program is going to, say encode a video, the period of time is probably the whole execution time of the task: it took 3h 21 minutes to encode the video and that results in a processing rate of 210.9 frames per second. For services that run indefinitely or when we're impatient, we might want to look at a shorter period of time and consider that. But if we choose the wrong period we'll miss important things: if we look at purchases per hour, if we look at Sunday night that might be misleading as compared to Monday at noon.

Look at the pretty pictures. Often, we want to not just see the raw data, but also the trend in the data; choosing the right visualization is really helpful. Digital speedometers in cars are more precise than sweep ones in terms of showing you the speed of your vehicle, but the sweeping analog speedometer makes it easier for a person to understand the acceleration (rate of change of that speed). Representing changes in data over time in a graph makes it easier to identify trends or alarming things, and as a bonus, managers and non-technical people love them and love to rely on them.

Consider just a simple example of the CPU usage graph from my laptop when I compile and run the nbody-parallel code; the blue represents user-space execution and red represents kernel execution:



It is obvious in this picture at what point the simulation starts running for real. If we see this and we can identify an operation that starts around the time the CPU is maxed out, we have found something useful.

Dashboards. If we put together enough pretty pictures, we get a... dashboard! Dashboards are (ideally) more than just pretty pictures of LINE GOES UP. It is instead intended to be a collection of useful graphs (time trends, histograms, etc) that present an easily-digestible summary of the situation. Good dashboards make it clear what's happening and what things need attention. Of course, dashboards tend to make more sense when we have many things to measure; not so much if we're just trying to optimize one specific thing.

Drill-down

All of this introduction to profiling has really been focused around the higher level ideas for identifying what's happening and what operations, if any, are slow (or slower than expected). Once we know that, for example, importing CSV files with tax rates is slow, we can start to use some profiling tools to identify what is slow inside that functionality. In the next topic, we'll start by narrowing down what area is the bottleneck.

25 — Load Testing

Load Testing

We've had a long look at the subject of observation—identifying areas in the execution of our program that are a potential issue. Now, we will also take some time to relate performance to scalability. Very early on in the course we mentioned the idea that what we want in scalability is to take our software from 1 user to 100 to 10 million. To scale up, we probably need to do some profiling to find out what's slow and make a decision about what to change. It's also useful in terms of making an estimate of our maximum number of users or transactions could be.

That's not a hypothetical scenario either; I [JZ] was asked by a C-Level (company executive) whether a particular system could handle $10\times$ as many users—and that answer had to be supported with some numbers. How would I make that determination? Analysis and testing, of course.

Start with why. The most important question when we want to start doing some load testing is: why are we doing this? A few possible answers [Mel21]:

- A new system is being developed, and management wants to understand the limits and the risks.
- The workload is expected to increase, and we (the CFO) want(s) to be sure the system can handle it.
- A sudden spike in workload is expected, like tax season, and we want to know we can handle it.
- The system has high uptime requirements, like 99.99%, and must work over an extended period of time.
- The project plan has a checkbox "performance testing", even though nobody has a real idea of what it means.

Leaving aside the last "reason", each of the actual reasons why implies a different kind of testing is required. If it's a new system, maybe we just need to understand the average workload (plus, perhaps, a buffer for safety) and make a plan for that. If the workload is expected to increase, like having ten times the number of users, we just need to establish our bottlenecks as in the next topic to identify what we think the limit is. If it's the spike situation, we need to find a way to generate that spike behaviour to see how the system responds, and the hardest part might actually be in how to create the test. If our reason is uptime requirements, then in addition to putting a lot of load on the system, we have to maintain it for an extended period to make sure there's no performance degradation—a test of endurance.

Stress Tests? Load testing is not the same thing as *stress testing*. Load testing involves having a specific target and the goal is to demonstrate that the application can handle that amount. Stress testing is about turning up the pressure (load) until things break[53], or at least stop working well. We are not going to go into stress testing in this topic, though it should be possible to take the load testing lessons and repurpose them by simply turning the setting up to even-bigger numbers.

Plans Are Useless, Planning Is Essential

The previously-cited book suggests making a plan that answers the questions of *who, what, where, when, & why*. We just covered the "why" question, and understood how it points us towards the answer of "what"—what kind of load testing are we intending to do here. That leaves "who" and "when". In a company (or FYDP) situation,

[53] The "ultimate breaking strain", as some say, or the ultimate tensile strength, in more modern language.

there might be reason for debate or discussion around who should do the tests and when they should take place. For our purposes the answers are: we are going to do them, and we're going to do them now.

How detailed the plan needs to be is going to depend on your organization, and the same applies for how many sign-offs (if any!) you need on the plan. Some companies have a high need for justification of the time invested in this, after all, because time is money (developer salary) and there are opportunity costs (what work are we not doing in favour of this). There's lots of literature out there about how to justify technical investments to senior management and let's not get sidetracked.

The load testing plan needs at least a definition of what will be tested, how it will be tested, and how we'll know if the test passes or fails. A simple scenario could look something like: we will test saving things to the database by generating 10 000 update-account requests, submit them all at once, and the the total time to complete all transactions should not exceed 15 seconds. That's all made up and probably even a little bit silly, but it meets the important criteria of explaining what to test, how to test it, and how to evaluate success. Let's expand on this.

What Workflows to Test?

While we might have a clear direction about the kind of test we want from why, the question still remains about what workflows are going to be tested. We cannot test everything—aiming for 100% coverage is unnecessary, but load testing should be reserved for only where it's truly needed because of its high cost and effort requirements.

If we already know which ones are slow (rate-limiting) or on the critical path, then we can certainly start there. Ideally, the observability/monitoring (you have it, right?) gives us the guidance needed here. If monitoring doesn't exist, that might need to get addressed first.

If it's a new (as-yet-unimplemented) system, which are the critical workflows for this application? Critical is determined by the nature of the product and what you want it to do. If something is computationally intensive, like compiling analytical data, then that workflow is important. In other scenarios, user experience determines what's critical for load testing: if we have information that says users quit the signup process if the signup page takes more than two seconds to load, we care about making sure the signup time doesn't exceed that limit even when the system is busy. There can also be external requirements that require load testing: if you are processing a transaction and you must return a yes or no response within 1 second, anything that is part of coming up with this answer is critical and worth testing.

If your current utilization is low, you might not know what the rate-limiting steps are at a glance. You can take a guess, but be prepared to revise those guesses partway through the process as you ramp up the load. Actually, you might need to do that even if utilization isn't low; you may find new things along the way that turn out to be the real limiting factors.

In the event of the uptime requirements, the tests likely look the same as the increased-workload situation—we just run them for longer. Endurance tests have significant overlap with load tests, but are not exactly the same. We'll come back to figuring out how long an endurance test should be shortly.

How to Test Them

Given a workflow and the kind of test that we want to do, then we just need to think about how to test it. Carrying out these tests might require provisioning additional (virtual) hardware, particularly if they are long-running. The typical test harness tools used for unit testing aren't always intended for this purpose, so you may need different tools, or just write a custom script to execute the tests.

If we want to run these tests to evaluate or experiment, or if we have an application developed and we want to estimate what its performance and limiting factors would be, there are some things we need to consider to make sure that the results we get are meaningful. So we should respect the following principles, as outlined in [Liu09].

Hardware Principle. Scalability testing is very different from QA testing (you test your code, right?) in that you will do development and QA on your local computer (or CI infrastructure) and all you really care about is whether the program produces the correct output "fast enough". That's fine, but it's no way to test scalability. If you actually want to test for scalability and real world performance, you have to do it on the machines that will

run the program in the live/production environment. Why? Well, low-end systems have very different limiting factors. You might be limited by the 16GB of RAM in your laptop—but the server might have 128GB of RAM. So you might spend a great deal of time worrying about RAM usage when it turns out it doesn't matter.

Reality Principle. It would be a good idea to use a "real" workload, as much as one can simulate. Legal reasons might prevent you from using actual customer data, but you should use the best approximation of it that you have.

If you only generate some test data, it might not be representative of the actual data: you might say 1% of your customers are on the annual subscription but if it's really 10% that might make a difference in how long it takes to run an analysis. On the other hand, your test data might be much larger than in production because your tests create (and don't delete) entities in the test system database every time you run.

This isn't theoretical: I [JZ] have been the incident responder on an incident where an app doesn't start up because the database migration takes too long and gets killed. This issue wasn't caught in the test environment because the size of data there was much smaller than the live database. So of course the migration was finished well under the time limit during testing... and failed when going live. Oops.

Related to that, user behaviour is hard to simulate, too. If you are making threads that pretend to be users to simulate concurrent usage of the system, the scripts those threads run may not accurately represent the way the users really behave. Your test example scenario may assume that company managers want to run the report once a month... your users might run them every hour. That one really happened to me [JZ] as well. The manager was using the report to keep an eye on what his team members were doing all day!

Volume Principle. "More is the new more." It's okay to use lighter workloads for regression testing and things like that, but if you actually want to see how your system performs under pressure, you actually need to put it under pressure! You can simulate pressure by limiting RAM or running something very CPU-intensive concurrently, but it's not quite the same as having an actually-high workload.

Why not? The issue isn't so much the test as it is what you're not seeing. If I want to test the app on my laptop under CPU pressure I can just run the app while I repeatedly convert a lecture video just to consume CPU cycles. That does have the effect of limited CPU time available and we can see how the app performs when CPU time is maxed out. But that's not the same as what happens if you have 500 users concurrently, because you might not get to full CPU usage with 500 users due to other reasons, such as lock contention.

These tests, incidentally, are of great interest to the customers and C-Levels, who would like to know that you can deliver promised levels of performance or throughput, or what the current maximum is.

Reproducibility and Regression Testing. Your results will need to be reproducible. That is, two different runs of the load testing on the same code should produce results that are very similar. Unlike unit tests where we expect results to be the same every time, load testing often has a larger degree of randomness and variance in execution because of the nature of the workload (e.g., randomly generate 10 000 customers) and the normal random factors caused by the operating system, scheduler, luck, and so on.

Endurance Tests: How Long?

Chances are you're familiar with the idea of endurance being different from peak workload. Can I [JZ] run at 10 km/h for 1 minute? Sure—I can run much faster than that for 1 minute! Can I run 30 minutes (5 km) at 10 km/h? Yes. 60 minutes (10 km)? Yes, but with difficulty. Four hours (40 km)? No, I'll get tired and slow down, stop, or maybe hurt myself. Okay, so we've found a limit here—I can endure an hour at this workload but things break down after that. If we only studied my running for less than half an hour, we might conclude that I could run at 10 km/h forever (or at least indefinitely), even though that's absolutely not true[54]. But if our sample period is 15 minutes, that is not long enough to reflect the cumulative negative effects that slow and stop me eventually.

Is this analogy suitable for software, though? CPUs don't get "tired", nor do their parts accumulate fatigue at

[54]That's a common fallacy in the media, too—you may notice, if you're looking for it, a plethora of journalists writing opinion pieces that say "the current situation is the new normal and will go on forever"—even though that's almost certainly not true. Things change all the time—political parties that win an election are often voted out again after a few years; rough job markets improve and good ones get tighter; high interest rates come down or low rates increase, etc. New and unprecedented things happen a lot, and change is constant.

anywhere near the same rate as a runner's muscles. Yes, it's still valid! A process that has a data hoarding problem is slowly accumulating memory usage and when its memory space is exhausted, we might encounter a hard stop (e.g., encountering the error `java.lang.OutOfMemoryError: Java heap space`) or just a degradation of performance based on the increasing amount of swapping memory to disk that is required. Same for filling up disk, exhausting file handles, log length, accumulated errors, whatever it is. That's a little bit like fatigue for the executing application, whether it's building in the program or its environment: it builds over time, and eventually the effects of it force a slowdown or stoppage of execution.

Accumulated "fatigue" for an application is not just a hypothetical, either. I [JZ] have personally seen services that encountered a problem due to running out of internal resources as a result of a holiday-season code freeze. That wasn't a load test in the sense of applying a higher load to validate execution rate—it was just an (unplanned surprise) endurance test. The solution to getting the error rate down involved restarting the instances one-by-one and things got back on track. This example calls back what I said about how endurance tests are not exactly the same as load tests, in that we can have an endurance test with low load, yet it may still be valid.

With that said, how do we identify what is the relevant period for the endurance test—or in the running analogy, how do I know if 30 minutes is the right running length to get a real idea? Should it be 3 hours? Once again, our first guide might be the product requirements for what we're building (testing) might give an idea. If it's an e-commerce platform then the endurance test might be something like five days to cover the period of US Thanksgiving, Black Friday, the weekend, and then Cyber Monday.

Other ideas for choosing endurance targets might consider the maintenance windows for the platform. Suppose you have a scheduled maintenance window that takes place on Sundays from 02:00–03:00 and there can be downtime during that period, according to the contracts (service level agreements) the company has signed. In that case, you want to validate that the system will work correctly and consistently long enough that it can be restarted (or updated) only during that maintenance window.

Unfortunately, there are no universal rules we can offer that give you the exact length of time to evaluate. You'll have to consider the requirements, the likely scenarios, and use your judgement.

How to Evaluate Success

There are two kinds of answers that load testing can give you (and they're related). The first is "Yes or no, can the system handle a load of X?"; the second is "What is the maximum load Y our current system can handle?". If we know Y then we can easily answer whether X is higher or lower. Between the two of them, this suggests we might prefer to find Y rather than answer the first question. But it might be hard to find the maximum limit. The difficulty of generating test data or load might increase much faster than the rate of load added to the system, and we might be crossing over into stress testing. Sometimes answering the first question is all that is necessary.

The value of Y may have some nuance above. The maximum rate we can handle may imply a hard stop—if the load exceeds Y then we crash, run out of memory, reject requests, or something else. It may also be a degradation of service: this is the point at which performance degrades below our target or minimum.

Observability has come up previously and it might have helped decide what's important. We might also need to add some monitoring or logging that tracks when events start and end, to gather data needed to make the overall evaluation. Examples that we are looking for are things like the following. Is the total work completed within the total time limit? Did individual items get completed within the item time limit 99% of the time or more?

As you might expect, the raw load test results are not always sufficient to make the call as to whether a test has passed or succeeded. Then there is post-processing to aggregate and analyze the data. Some manual work might also be necessary to correlate the data with other known factors, particularly if it had not been possible to test on separate hardware or separate instances [Mel21]. At the end of this process, hopefully you can look at the outcomes and conclude whether a given scenario is passed or failed.

Finally, success in an endurance test is not just answering whether the system was able handle a load of X, but is about whether that answer continued to be yes, consistently, across a long period of time. In an endurance test especially, looking at the trend may be more instructive as to how well things are working.

So You Failed the Load Test...

Like a software program, when it comes to running, I [JZ] can get better—I just need to make some changes. To get better at endurance running, chances are I mostly just need to increase (slowly) the running distances and practice more and I'll get better. Or as the internet might say, git gud (get good).

The idea works for the programs too—if the load test has failed, we'll have one or more specific scenarios to improve and we can apply techniques from this course (or elsewhere) to make that part better. Then re-run the test and re-evaluate. If all is well, done; otherwise, repeat as necessary until we've passed the scenario.

And also like a software program, there is a point at which I can make no more improvements. At the time of writing, Google says that the world record for marathon running is 2:01:09, held by Eliud Kipchoge, or an average speed of 21.02 km/h. There is absolutely no chance I can get to that level, no matter how hard I train. So I am not getting selected for the Canadian Olympic team.

If we've reached the limits of what we can do in the software, it might not be the right tool for your needs and a major redesign or replacement is needed. Designing the system with higher load in mind is sometimes possible, though even in situations where it's possible, the cost might be prohibitive. Are we stuck?

No! You can have different expectations. A four-hour marathon seems achievable for me if I worked on it. If it's unrealistic to bill all your customers on the same day, why not convince the company to let billing be spread across the whole month? Think about the problem outside the constraints that are currently given.

Constant Vigilance

You may have heard the saying "constant vigilance" around the topic of defending against the dark arts. Load testing at a given time captures the state of things at that time only. It needs to be repeated regularly to catch performance degradations that would make your software fail to meet its targets or design load. These are rarely intentional. But, software grows in complexity and in functionality, both of which are likely to make it slower. Improved hardware does, over time, offset some of the slowdown of added complexity. However, at the time of writing, this is an era of small, incremental improvements year-to-year, not big leaps and bounds forward [55]. So, for now, we must still repeat the load tests often enough to catch major regressions before they become a problem.

Real-life example: `https://arewefastyet.com` tracks regressions/improvements in Firefox performance.

[55] Considering the previous footnote, it would be thoroughly foolish to now fall into the trap of saying "the current situation will go on forever" and say that there will never be revolutionary change in execution hardware that offsets the increases in complexity. But that's not where we are right now.

26 — Finding Bottleneck Devices

Characterizing Performance & Scalability Problems

Studies show that poor mobile app performance, whether due to high usage of resources (e.g., battery, memory) or to just being slow, is a major complaint that users write about in app stores [KSNH15]. The paper is somewhat older at this point, and while we might like to think that developers are doing a better job of proactively identifying issues, chances are it's still the case that a distressing number of issues are first discovered when someone posts a negative review. *"One star, because it's not possible to give zero!"*.

A user might be able to give a vague idea of what's wrong, or point out a workflow where they're dissatisfied with the performance. Then it's up to the development team to figure out what's the cause of the problem. Most of our next topics in profiling will be around the idea of CPU profiling, which is to say, examining where CPU time is going. That's generally predicated on the assumption that CPU time is limiting factor. But maybe it isn't, and before we go about focusing on how to examine CPU usage, let's check that assumption—because it could be something else.

> *It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.*
>
> - Sherlock Holmes (*A Scandal in Bohemia*; Sir Arthur Conan Doyle)

Keeping the wisdom of Mr. Holmes in mind, we need to collect evidence before reaching conclusions. At a high level we probably have the following potential culprits to start with:

1. CPU
2. Memory
3. Disk
4. Network
5. Locks

Caveats. The list above is, obviously, just categories, but they are starting points for further investigation. They are listed in some numerical order, which we'll follow for convenience, but there is no reason why one would have to investigate them in that order.

Even if we identify a culprit, fixing it is a separate issue involving the application of the techniques covered elsewhere in the course. That might be really difficult—sometimes the cause of the problem is simply that the user's device/hardware is too old (or lacking important hardware for acceleration). Games are likely the most obvious example of situations where the "minimum" and "recommended" hardware configurations are published and if you don't meet those requirements, you're going to have a bad time (if the game runs at all). That applies to apps too, though dropping older devices from the supported set is more likely a result of the device no longer getting OS/API upgrades rather than being "too slow". All of which is to say, at the end of the analysis, sometimes the correct solution is to advise the user to upgrade their hardware.

Another possible outcome of finding the bottleneck when reported by the user is that it actually isn't a performance problem as much as a programming error. For example, the user may report the GUI lagging or the application not

responding (up to and including the actual system not-responding dialog), but these scenarios are not the result of low computational power on the device; they are caused instead by doing some slow or computationally-intensive task in the UI thread rather than in the background [LVVLP15]. The fix there is just to, well, fix the bug.

CPU. CPU is probably the easiest of these to diagnose. Something like `top` or Task Manager will tell you pretty quickly if the CPU is busy. You can look at the %CPU columns and see where all your CPU is going. Still, that tells you about right now; what about the long term average? Checking with my machine "Loki", that used to donate its free CPU cycles to world community grid (I was singlehandedly saving the world, you see. I mean. I did stop in 2016, and look at what's happened since then):

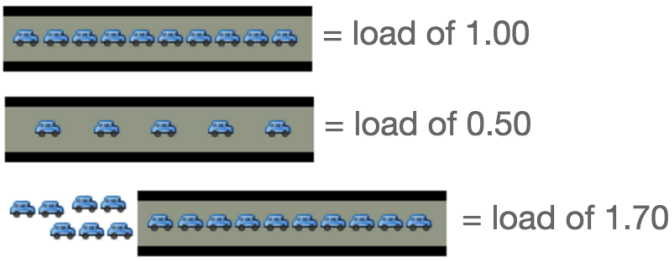
```
top - 07:28:19 up 151 days, 23:38,  8 users,  load average: 0.87, 0.92, 0.91
```

Those last three numbers are the one, five, and fifteen minute averages of CPU load, respectively. Lower numbers mean less CPU usage and a less busy machine. A small guide on how to interpret this, from [And15].

Picture a single core of a CPU as a lane of traffic. You are a bridge operator and so you need to monitor how many cars are waiting to cross that bridge. If no cars are waiting, traffic is good and drivers are happy. If there is a backup of cars, then there will be delays. Our numbering scheme corresponds to this:

1. 0.00 means no traffic (and in fact anything between 0.00 and 0.99); means we're under capacity and there will be no delay.
2. 1.00 means we are exactly at capacity. Everything is okay, but if one more car shows up, there will be a delay.
3. Anything above 1.00 means there's a backup (delay). If we have 2.00 load, then the bridge is full and there's an equal number of cars waiting to get on the bridge.

Or, visually, also from [And15]:



Being at or above 1.00 isn't necessarily bad, but you should be concerned if there is consistent load of 1.00 or above. And if you are below 1.00 but getting close to it, you know how much room you have to scale things up—if load is 0.4 you can increase handily. If load is 0.9 you're pushing the limit already. If load is above 0.70 then it's probably time to investigate. If it's at 1.00 consistently we have a serious problem. If it's up to 5.00 then this is a red alert situation.

Now this is for a single CPU—if you have a load of 3.00 and a quad core CPU, this is okay. You have, in the traffic analogy, four lanes of traffic, of which 3 are being used to capacity. So we have a fourth lane free and it's as if we're at 75% utilization on a single CPU.

Memory and Disk. Next on the list is memory. If you are using some garbage-collected language or framework, you will find lots of runs of the garbage collector, or at least very long ones when it does run; in the worst case scenario you'll see your application run out of memory and either crash or recover from it [LVVLP15].

One way to tell if memory is the limiting factor is actually to look at disk utilization. If there is not enough RAM in the system, there will be swapping and then performance goes out the window and scalability goes with. That is of course, the worst case. You can ask via `top` about how much swap is being used, but that's probably not the interesting value.

```
KiB Mem:   8167736 total,  6754408 used,  1413328 free,   172256 buffers
KiB Swap:  8378364 total,  1313972 used,  7064392 free.  2084336 cached Mem
```

This can be misleading, because memory being "full" does not necessarily mean anything bad. It means the resource is being used to its maximum potential, yes, but there is no benefit to keeping a block of memory open for no reason. Things will move into and out of memory as they need to, and nobody hands out medals to indicate that you did an awesome job of keeping free memory. It's not like going under budget in your department for the year. Also, memory is not like the CPU; if there's nothing for the CPU to do, it will just idle (or go to a low power state, which is nice for saving the planet). But memory won't "forget" data if it doesn't happen to be needed right now - data will hang around in memory until there is a reason to move or change it. So freaking out about memory appearing as full is kind of like getting all in a knot about how "System Idle Process" is hammering the CPU[56].

You can also ask about page faults, with the command `ps -eo min_flt,maj_flt,cmd` which will give you the major page faults (had to fetch from disk) and minor page faults (had to copy a page from another process). The output of this is too big even for the notes, but try it yourself (or I might be able to do a demo of it in class). But this is lifetime and you could have a trillion page faults at the beginning of your program and then after that everything is fine. What you really want is to ask Linux for a report on swapping:

```
jz@Loki:~$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0 1313972 1414600 172232 2084296    0    0     3    39    1    1 27  1 72  0  0
 0  0 1313972 1414476 172232 2084296    0    0     0    21  359  735 19  0 80  0  0
 0  0 1313972 1414656 172236 2084228    0    0     0   102  388  758 22  0 78  0  0
 4  0 1313972 1414592 172240 2084292    0    0     0    16  501  847 33  0 67  0  0
 0  0 1313972 1412028 172240 2084296    0    0     0     0  459  814 29  0 71  0  0
```

In particular, the columns "si" (swap in) and "so" (swap out) are the ones to pay attention to. In the above example, they are all zero. That is excellent and tends to indicate that we are not swapping to disk and that's not the performance limiting factor. Sometimes we don't get that situation. A little bit of swapping may be inevitable, but if we have lots of swapping, we have a very big problem. Here's a not-so-nice example, from [Tan05]:

```
  procs                  memory    swap          io     system cpu
r  b  w   swpd   free  buff cache  si  so   bi   bo   in    cs us  sy  id
. . .
1  0  0  13344   1444  1308 19692   0 168  129   42 1505   713 20  11  69
1  0  0  13856   1640  1308 18524  64 516  379  129 4341   646 24  34  42
3  0  0  13856   1084  1308 18316  56  64   14    0  320  1022 84   9   8
```

If we're not doing significant swapping, then memory isn't holding us back, so we can conclude it is not the limiting factor in scaling the application up. On to disk.

Looking at disk might seem slightly redundant if memory is not the limiting factor. After all, if the data were in memory it would be unnecessary to go to disk in the first place. Still, sometimes we can take a look at the disk and see if that is our bottleneck.

```
jz@Loki:~$ iostat -dx /dev/sda 5
Linux 3.13.0-24-generic (Loki)  16-02-13  _x86_64_ (4 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.24     2.78    0.45    2.40    11.60   154.98   116.91     0.17   61.07   11.57   70.27   4.70   1.34
```

It's that last column, %util, that tells us what we want to know. The device bandwidth here is barely being used at all. If you saw it up at 100% then you would know that the disk was being maxed out and that would be a pretty obvious indicator that it is the limiting factor. This does not tell you much about what is using the CPU, of course, and you can look at what processes are using the I/O subsystems with `iotop` (requires root privileges[57]).

[56] Yes, a tech journalist named John Dvorak really wrote an article about this, and here I am roasting him about it decades later because it's just so ridiculous a theory that I can't help it.

[57] `https://xkcd.com/149/`

Network. That leaves us with networks. We can ask about the network with `nload`, which gives the current, average, min, max, and total values. And you get a nice little graph if there is anything to see. It's not so much fun if nothing is happening. But you'll get the summary, at least:

```
Curr: 3.32 kBit/s
Avg: 2.95 kBit/s
Min: 1.02 kBit/s
Max: 12.60 kBit/s
Ttl: 39.76 GByte
```

So if you saw here, for example, that data was leaving at 100 Megabits per second you'd have a pretty good idea that was the limitation, but you may still be network limited at lower speeds. Intermediary devices or other non-optimal hardware can get in the way. For example, it's possible to run a wired network over power lines—this is not optimal, but it's effective in older buildings where it would be difficult or expensive to run network cables through the walls. This will limit your speed based on the condition of the wires in the wall, alongside any other devices on the same circuit adding noise to the signal. So what should be 1000 or 100 MBit might actually only be more like 32 Mbit. Wireless networks have the same problem, being affected by walls, floors, electromagnetic interference, humidity...

Testing network speed can be done using tools like speedtest.net, which gives some indication of what the network connection speed is like from a given device (upload and download). You may need to test multiple times to get a realistic picture of speed. The test validates the speed of connection to the upstream system (e.g., some server operated by the speed test service), but not necessarily to your own data centre. Both locations can have good network connections to the outside world, but might not be easily able to talk to each other well.

In my [JZ] experience, I've seen network speed issues for people working in Hong Kong using an application that has the backend deployed in a Frankfurt (Germany) data centre. The problem here is not bandwidth but latency. Tools like speedtest can tell you the latency of communication alongside the bandwidth. If you want to get an idea of the path and the latency to a particular remote system, you can use the `traceroute` tool. Here is an example from Catchpoint `https://www.catchpoint.com/network-admin-guide/how-to-read-a-traceroute`, which also provides some guidance on how to interpret a traceroute result:

```
Microsoft Windows [Version 10.0.19043.1288]
(c) Microsoft Corporation. All rights reserved.
C:\Users\Michael>tracert catchpoint.com
Tracing route to catchpoint.com [64.79.149.76]
Over a maximum of 30 hops:
1 2ms 1ms 1ms 10.0.0.1
2  10ms 10ms 10ms 96.120.40.245
3 10ms 11ms 12ms 96.110.175.85
4 10ms 16ms 10ms  162.151.63.57
5 19ms 16ms 20ms 96.108.21.57
6 15ms 19ms 14ms 96.216.134.10
7           19ms  22ms  21ms  be-32121-cs02.350ecermak.il.ibone.comcast.net [96.110.42.181]
8           22ms  34ms  22ms  be-2204-pe04.350ecermak.il.ibone.comcast.net [96.110.37.38]
9           22ms  20ms  20ms  50.208.234.106
10        51ms  50ms  49ms  ae18-0.cr02.dlls02-tx.us.windstream.net [40.128.10.135]
11        73ms  72ms  72ms  ae4-0.agr03.phnd01-az.us.windstream.net [169.130.193.231]
12        84ms  73ms  75ms  ae1-0.pe05.phnd01-az.us.windstream.net [169.130.169.31]
13        85ms  84ms  85ms  h241.23.132.40.static.ip.windstream.net [40.132.23.241]
14        *          82ms  78ms  be181.las-n10s1-core1.switch.com [66.209.64.121]
15        79ms  77ms  80ms  bell011.las-agg7s5-1.switch.com [66.209.72.26]
16        79ms  77ms  79ms  64.79.139.18
17        77ms  77ms  87ms  64.19.149.76
Trace complete
```

Remember that communication latency can never be truly eliminated, because it takes non-zero time for packets to get processed through network hardware (e.g., switches), and, more importantly, the speed of light. I looked up some data on `https://wondernetwork.com/pings/New%20York`—use with a grain of salt—that says the ping from New York to Lyon is 73.21ms which is something like 83.79% of the speed of light in fibre-optic cable (as of August 2024). Even if we got it up to 100% of the speed of light in fibre-optic cable, or used some other material that had a higher speed of light in it, it can't ever get down to nothing.

One more thing that can cut into your effective bandwidth is packet loss: data getting dropped or corrupted en route. That requires some device along the line to identify that the packets are not as they should be, re-request the needed packets, and wait for them to arrive. Packet loss may be environmental, but it also might mean a device needs replacing.

Locks. The last possibility we'll consider is that your code is slow because we're waiting for locks, either frequently or for lengthy periods of time. We've already discussed appropriate use of locks, so we won't repeat that. The discussion here is about how to tell if there is a locking problem in the first place.

We'll exclude the discussion of detecting deadlock, because we'll say that deadlock is a correctness problem more than a performance problem. In any case, a previous course (ECE 252, SE 350, MTE 241) very likely covered the idea of deadlock and how to avoid it. The Helgrind tool (Valgrind suite) is a good way to identify things like lock ordering problems that cause a deadlock. Onwards then.

Unexpectedly low CPU usage, not explained by I/O-waits, may be a good indicator of lock contention. If that's the case, when CPU usage is low we would see many threads are blocked.

Unlike some of the other things we've discussed, there's no magical `locktrace` tool that would tell us about critical section locks, and the POSIX pthread library does not have any locks tracing in its specification [Sit21]. One possibility is to introduce some logging or tracing ourselves, e.g., recording in the log that we want to enter a critical section A and then another entry once we're in it and a third entry when we leave A. That's not great, but it is something!

I did some reading about `perf lock` but the problem is, as above, that it doesn't really find user-space lock contention. Tools can tell you about thread switches but that's not quite the same. Other commercial tools like Intel VTune claim that they can find these sorts of problems. But those cost money and may be CPU-vendor-specific.

But it's probably CPU...

Most profiling tools, and most of our discussion, will be about CPU profiling. It's the most likely problem we'll face and something that we are hopefully able to do something about.

27 — Program Profiling and POGO

Profiling

We've spent some time talking about observing programs at a higher level, but now we should consider zooming in a little bit to using specific profiling tools to figure out where CPU time is going and how to make use of that.

I've used the logging-only approach myself to figure out what blocks of a single large function are taking a long time (such as a time when I found out that updating exchange rates was slow.). But this approach is not necessarily a good one. It's an example of "invasive" profiling—we are going in and changing the source code of the program in question—to add (slow!) instrumentation (log/debug statements). Plus we have to do a lot of manual accounting. Assuming your program is fast and goes through functions quickly and often, trying to put the pieces together manually is hopeless. It worked in that one example because the single function itself was running in the half-hour range and I could see that the save operation was taking twelve minutes. Not kidding.

I also mentioned earlier that I used the profiling tool for CUDA to find out what was wrong with my N-Body program. I ran the command `nvprof target/release/nbody-cuda`, and in addition to the regular program output I got the following, which showed that the time was going to the kernel and I wasn't losing a lot in overhead:

```
==20734== Profiling application: target/release/nbody-cuda
==20734== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  10.7599s         1  10.7599s  10.7599s  10.7599s  calculate_forces
                    0.00%  234.72us         2  117.36us  100.80us  133.92us  [CUDA memcpy HtoD]
                    0.00%  94.241us         1  94.241us  94.241us  94.241us  [CUDA memcpy DtoH]
      API calls:   97.48%  10.7599s         1  10.7599s  10.7599s  10.7599s  cuStreamSynchronize
                    1.92%  211.87ms         1  211.87ms  211.87ms  211.87ms  cuCtxCreate
                    0.54%  59.648ms         1  59.648ms  59.648ms  59.648ms  cuCtxDestroy
                    0.04%  4.8704ms         1  4.8704ms  4.8704ms  4.8704ms  cuModuleLoadData
                    0.00%  404.72us         2  202.36us  194.51us  210.21us  cuMemAlloc
                    0.00%  400.58us         2  200.29us  158.08us  242.50us  cuMemcpyHtoD
                    0.00%  299.30us         2  149.65us  121.42us  177.88us  cuMemFree
                    0.00%  243.86us         1  243.86us  243.86us  243.86us  cuMemcpyDtoH
                    0.00%  85.000us         1  85.000us  85.000us  85.000us  cuModuleUnload
                    0.00%  41.356us         1  41.356us  41.356us  41.356us  cuLaunchKernel
                    0.00%  18.483us         1  18.483us  18.483us  18.483us  cuStreamCreateWithPriority
                    0.00%  9.0780us         1  9.0780us  9.0780us  9.0780us  cuStreamDestroy
                    0.00%  2.2080us         2  1.1040us     215ns  1.9930us  cuDeviceGetCount
                    0.00%  1.4600us         1  1.4600us  1.4600us  1.4600us  cuModuleGetFunction
                    0.00%  1.1810us         2     590ns     214ns     967ns  cuDeviceGet
                    0.00%     929ns         3     309ns     230ns     469ns  cuDeviceGetAttribute
```

Oh, and for comparison, here's the one where I make much better use of the GPU's capabilities (with better grid and block settings):

```
=22619== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.92%  417.53ms         1  417.53ms  417.53ms  417.53ms  calculate_forces
                    0.06%  236.03us         2  118.02us  101.44us  134.59us  [CUDA memcpy HtoD]
                    0.02%  93.057us         1  93.057us  93.057us  93.057us  [CUDA memcpy DtoH]
      API calls:   52.09%  417.54ms         1  417.54ms  417.54ms  417.54ms  cuStreamSynchronize
                   26.70%  214.00ms         1  214.00ms  214.00ms  214.00ms  cuCtxCreate
                   13.63%  109.26ms         1  109.26ms  109.26ms  109.26ms  cuModuleLoadData
                    7.42%  59.502ms         1  59.502ms  59.502ms  59.502ms  cuCtxDestroy
                    0.05%  364.08us         2  182.04us  147.65us  216.42us  cuMemcpyHtoD
                    0.04%  306.48us         2  153.24us  134.10us  172.37us  cuMemAlloc
                    0.04%  285.73us         2  142.86us  122.90us  162.83us  cuMemFree
```

```
0.03%  246.37us         1  246.37us  246.37us  246.37us  cuMemcpyDtoH
0.01%  61.916us         1  61.916us  61.916us  61.916us  cuModuleUnload
0.00%  26.218us         1  26.218us  26.218us  26.218us  cuLaunchKernel
0.00%  15.902us         1  15.902us  15.902us  15.902us  cuStreamCreateWithPriority
0.00%  9.0760us         1  9.0760us  9.0760us  9.0760us  cuStreamDestroy
0.00%  1.6720us         2     836ns     203ns  1.4690us  cuDeviceGetCount
0.00%  1.0950us         1  1.0950us  1.0950us  1.0950us  cuModuleGetFunction
0.00%     888ns         3     296ns     222ns     442ns  cuDeviceGetAttribute
0.00%     712ns         2     356ns     212ns     500ns  cuDeviceGet
```

Nicholas Nethercote wrote the "counts" tool[58] to process ad-hoc debug output and shows an example of using it to profile the size of heap allocations in Firefox. In the example in the README, he reports that while most allocations are small, most memory is allocated as part of a large allocation. He provides a number of other examples where ad-hoc profiling is useful: how often are certain paths executed? how many times does a loop iterate? how many elements are in a hash table at a given location? And more! It's hard to write a general-purpose tool for these ad-hoc queries.

(Also like debugging, if you get to be a wizard you can maybe do it by code inspection, but that technique of speculative execution inside your head is a lot harder to apply to performance problems than it is to debugging. Trained professionals like Nicholas Nethercote use profilers, so you can too.)

So we should all agree, we want to use tools and do this in a methodical way.

Now that we agree on that, let's think about how profiling tools work

- sampling-based (traditional): every so often (e.g. 100ms for gprof), stop the system and ask it what it's doing (query the system state); or,
- instrumentation-based, or probe-based/predicate-based (sometimes, too expensive): query system state under certain conditions; like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling. You can read more about profiling in Chapter 5 (Profiling) of The Rust Performance Book [N+20]; it discusses heap profiling in much more detail than we do (or don't, more accurately).

If you need your system to run fast, you need to start profiling and benchmarking as soon as you can run the system. Benefits:

- establishes a baseline performance for the system;
- allows you to measure impacts of changes and further system development;
- allows you to re-design the system before it's too late;
- avoids the need for "perf spray" to make the system faster, since that spray is often made of "unobtainium"[59].

Tips for Leveraging Profiling. When writing large software projects:

- First, write clear and concise code. Don't do any premature optimizations—focus on correctness. Still, there are choices you can make that support performance at this stage, like using an efficient search or sort algorithm, if you know it's better and won't take additional effort.
- Once it's working, profile to get a baseline of your performance: it allows you to easily track any performance changes and to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Look for abnormalities; in particular, you're looking for deviations from the following rules:

- time is spent in the right part of the system/program;

[58] https://github.com/nnethercote/counts
[59] http://en.wikipedia.org/wiki/Unobtainium

- time is not spent in error-handling, noncritical code, or exceptional cases; and
- time is not unnecessarily spent in the operating system.

For instance, "why is `ps` taking up all my cycles?"; see page 34 of [Can06].

Development vs. production. You can always profile your systems in development, but that might not help with complexities in production. (You want separate dev and production systems, of course!) We'll talk a bit about DTrace, which is one way of profiling a production system. The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Userspace per-process profiling

Sometimes—or, in this course, often—you can get away with investigating just one process and get useful results about that process's behaviour. We'll first talk about `perf`, the profiler recommended for use with Rust. This is Linux-specific, though.

The `perf` tool is an interface to the Linux kernel's built-in sample-based profiling using CPU counters. It works per-process, per-CPU, or system-wide. It can report the cost of each line of code.

Webpage: `https://perf.wiki.kernel.org/index.php/Tutorial`

Here's a usage example on some old assignment code from a previous offering of the course:

```
[plam@lynch nm-morph]$ perf stat ./test_harness

 Performance counter stats for './test_harness':

       6562.501429 task-clock                #    0.997 CPUs utilized
               666 context-switches          #    0.101 K/sec
                 0 cpu-migrations            #    0.000 K/sec
             3,791 page-faults               #    0.578 K/sec
    24,874,267,078 cycles                    #    3.790 GHz                      [83.32%]
    12,565,457,337 stalled-cycles-frontend   #   50.52% frontend cycles idle     [83.31%]
     5,874,853,028 stalled-cycles-backend    #   23.62% backend  cycles idle     [66.63%]
    33,787,408,650 instructions              #    1.36  insns per cycle
                                             #    0.37  stalled cycles per insn [83.32%]
     5,271,501,213 branches                  #  803.276 M/sec                    [83.38%]
       155,568,356 branch-misses             #    2.95% of all branches          [83.36%]

       6.580225847 seconds time elapsed
```

Right, let's get started. We're going to use the blog post [Per16] as a guide; that source tells a more complete story of an example of using the profiler to optimize, but for now we are just interested in the steps.

The first thing to do is to compile with debugging info, go to your `Cargo.toml` file and add:

```
[profile.release]
debug = true
```

This means that `cargo build -release` will now compile the version with debug info (you can tell because it will say `Finished release [optimized + debuginfo] target(s) in 0.55s`; without this, we wouldn't get the part that says debug info so we can tell if it's correct. And we want it to be the release version that we're instrumenting, because we want the compiler optimizations to be applied. Without those, we might be trying to optimize things where the compiler would do it for us anyway.

The basic plan is to run the program using `perf record`, which will sample the execution of the program to produce a data set. Then there are three ways we can look at the code: `perf report`, `perf annotate`, and a flamegraph. We'll look at all of those, but in a live demo.

CLion. While we've seen how to use `perf`, it's not the only way. During development of some of the code exercises, I used the CLion built-in profiler for this purpose. It generates a flamegraph for you too, and I'll show that for how to create the flamegraph as well.

Profiler Guided Optimization (POGO)

A few years ago, we were fortunate enough to have a guest lecture from someone at Microsoft actually in the room to give the guest lecture on the subject of Profile Guided Optimization (or POGO). In subsequent years, I was not able to convince him to fly in just for the lecture. Anyway, let's talk about the subject, which is by no means restricted to Rust.

The compiler does static analysis of the code you've written and makes its best guesses about what is likely to happen. The canonical example for this is branch prediction: there is an if-else block and the compiler will then guess about which is more likely and optimize for that version. Consider three examples, originally from [Ast13a] but replaced with some Rust equivalents:

```
fn which_branch(a: i32, b: i32) {
    if a < b {
        println!("Case one.");
    } else {
        println!("Case two.");
    }
}
```

Just looking at this, which is more likely, `a < b` or `a >= b`? Assuming there's no other information in the system, the compiler can believe that one is more likely than the other, or having no real information, use a fallback rule. This works, but what if we are wrong? Suppose the compiler decides it is likely that `a` is the larger value and it optimizes for that version. However, it is only the case 5% of the time, so most of the time the prediction is wrong. That's unpleasant. But the only way to know is to actually run the program.

```
trait Polite {
    fn greet(&self) -> String;
}

struct Kenobi {
    /* Stuff */
}
impl Polite for Kenobi {
    fn greet(&self) -> String {
        return String::from("Hello there!");
    }
}
```

```
struct Grievous {
    /* Things */
}
impl Polite for Grievous {
    fn greet(&self) -> String {
        return String::from("General Kenobi.");
    }
}

fn devirtualization(thing: &Polite) {
    println!("{}", thing.greet());
}
```

There are similar questions raised for the other two examples. What is the "normal" type for some reference `thing`? It could be of either type `Kenobi` or `Grievous`. If we do not know, the compiler cannot do devirtualization (replace this virtual call with a real one). If there was exactly one type that implements the `Polite` trait we wouldn't have to guess. But are we much more likely to see `Kenobi` than `Grievous`?

```
fn match_thing(x: i32) -> i32 {
    match x {
        0..10 => 1,
        11..100 => 2,
        _ => 0
    }
}
```

Same thing with `x`: what is its typical value? If we know that, it is our prediction. Actually, in a match block with many options, could we rank them in descending order of likelihood?

There exists a solution to this, and it is that we can give hints to the compiler, but that's a manual process. Automation is a good thing and this lecture is about that. These sorts of things already exist for Java! The Java HotSpot virtual machine will update its predictions on the fly. There are some initial predictions and if they turn out to be wrong, the Just In Time compiler will replace it with the other version. That's neat! I don't know for certain but I suspect the .NET runtime will do the same for something like C#. But this is Rust and we don't have the runtime to reduce the overhead: the compiler runs and it does its job and that's it; the program is never updated with newer predictions if more data becomes known.

Solving this problem is the goal of POGO. It is taking the data from some actual runs of the program and using that to inform the predictions. This necessitates a multi-step compile: first compile the code, run it to collect data, then recompile the code using the data we collected. Let's expand on all three steps.

Step one is to generate an executable with instrumentation. Ask to compile with instrumentation enabled, which also says what directory to put it in: `-Cprofile-generate=/tmp/pgo-data`. The compiler inserts a bunch of probes into the generated code that are used to record data. Three types of probe are inserted: function entry probes, edge probes, and value probes. A function entry probe, obviously, counts how many times a particular function is called. An edge probe is used to count the transitions (which tells us whether an if branch is taken or the else condition). Value probes are interesting; they are used to collect a histogram of values. Thus, we can have a small table that tells us the frequency of what is given in to a `match` statement. When this phase is complete, there is an instrumented executable and an empty database file where the training data goes [Ast13a].

Step two is training day: run the instrumented executable through real-world scenarios. Ideally you will spend the training time on the performance-critical sections. It does not have to be a single training run, of course. Data can be collected from as many runs as desired. Keep in mind that the program will run a lot slower when there's the instrumentation present.

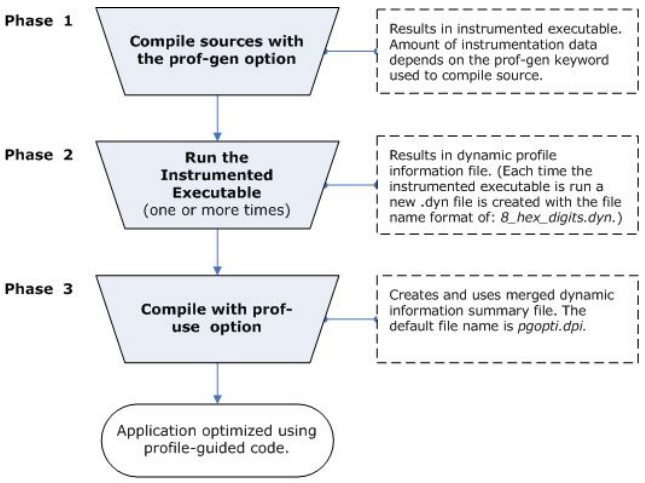
Still, it is important to note that you are not trying to exercise every part of the program (this is not unit testing); instead it should be as close to real-world-usage as can be accomplished. In fact, trying to use every bell and whistle of the program is counterproductive; if the usage data does not match real world scenarios then the compiler has been given the wrong information about what is important. Or you might end up teaching it that almost nothing is important...

According to the docs about it[60], the output `.profraw` files require a little bit of processing before they're ready to go. When the program is running, the recording of data is done as quickly as possible with little regard for making it neat. Think of it like taking notes furiously during a lecture and then later revisiting them to organize them a bit. The tool for doing this is `llvm-profdata` and it will organize the data into a `.profdata` file. We can merge multiple runs as needed into a single file that will be used for input.

Step three is a recompile. This time, in addition to the source files, the (merged) training data is fed into the compiler for a second compile, and this data is applied to (hypothetically) produce a better output executable than could be achieved by static analysis alone.

It is not necessary to do all three steps for every build. Old training data can be re-used until the code base has diverged significantly enough from the instrumented version. According to [Ast13a], the recommended workflow is for one developer to perform these steps and check the training data into source control so that other developers can make use of it in their builds.

The Intel Developer Zone explains the process in a handy infographic[61] :



[60] `https://doc.rust-lang.org/rustc/profile-guided-optimization.html`
[61] Source: `https://software.intel.com/en-us/node/522721`

Or, here, a complete set of steps for actually running it if our program is all in `main.rs`, from the docs[62]:

```
# STEP 1: Compile the binary with instrumentation
rustc -Cprofile-generate=/tmp/pgo-data -O ./main.rs

# STEP 2: Run the binary a few times, maybe with common sets of args.
#         Each run will create or update '.profraw' files in /tmp/pgo-data
./main mydata1.csv
./main mydata2.csv
./main mydata3.csv

# STEP 3: Merge and post-process all the '.profraw' files in /tmp/pgo-data
llvm-profdata merge -o ./merged.profdata /tmp/pgo-data

# STEP 4: Use the merged '.profdata' file during optimization. All 'rustc'
#         flags have to be the same.
rustc -Cprofile-use=./merged.profdata -O ./main.rs
```

(NB: Debian/Ubuntu's rustc doesn't seem to support PGO; I had to rustup it myself on my computer; I also used the Complete Cargo Workflow listed later.)

What does it mean for the executable to be better? We have already looked at an example about how to predict branches. Predicting it correctly will be faster than predicting it incorrectly, but this is not the only thing. The algorithms will aim for speed in the areas that are "hot" (performance critical and/or common scenarios). The algorithms will alternatively aim to minimize the size of code of areas that are "cold" (not heavily used). It is recommended in [Ast13a] that less than 5% of methods should be compiled for speed.

It is possible that we can combine multiple training runs and we can manually give some suggestions of what scenarios are important. The more a scenario runs in the training data, the more important it will be, as far as the POGO optimization routine is concerned, but also, multiple runs can be merged with user assigned weightings.

Behind the Scenes

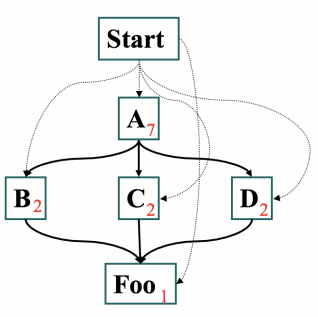
In the optimize phase, the training data is used to do the following optimizations (which I will point out are based on C and C++ programs and not necessarily Rust, but the principles should work because the Rust compiler's approach to this is based on that of LLVM/Clang) [Ast13b]:

1. Full and partial inlining
2. Function layout
3. Speed and size decision
4. Basic block layout
5. Code separation
6. Virtual call speculation
7. Switch expansion
8. Data separation
9. Loop unrolling

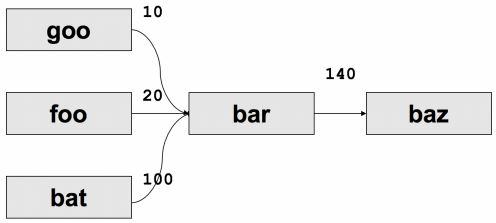
For the most part we should be familiar with the techniques that are listed as being other compiler optimizations we have previously discussed. The new ones are (3) speed and size decision, which we have just covered; and items (4) and (5) which relate to how to pack the generated code in the binary.

According to [Ast13b] the majority of the performance gains relate to the inlining decisions. These decisions are based on the call graph path profiling: the behaviour of function `foo` may be very different when calling it from `bar` than it is when calling it from function `baz`. Let's look at this call graph from [Ast13b]:

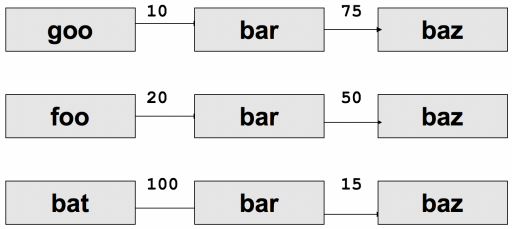
[62] `https://doc.rust-lang.org/rustc/profile-guided-optimization.html`



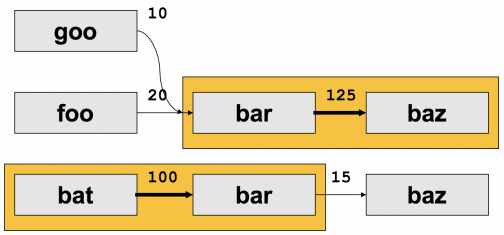
Quick analysis of this code would have us find all the ways in which the functions might call each other. In total, there are 14 paths in this code, seven of which get us to function `Foo`. Consider another diagram showing the relationships between functions, in which the numbers on the edges represent the number of invocations [Ast13b]:



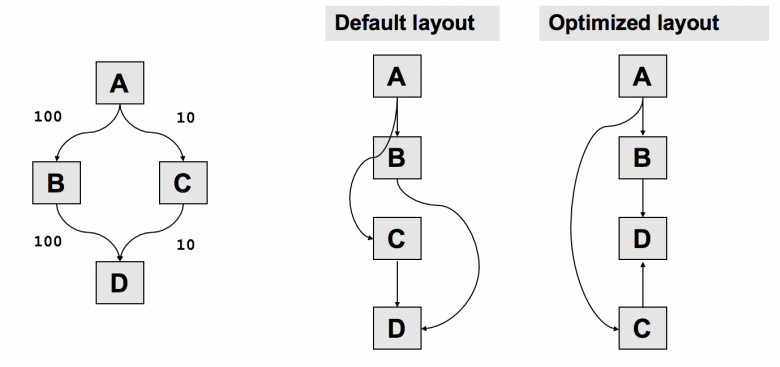
When considering what to do here, POGO takes the view like this [Ast13b]:



Each part of the call path is considered separately, remembering that we want to inline where it makes sense for speed, but otherwise leave it alone because of code size increases. Inlining `bar` into `bat` makes sense, but not inlining `bar` into `goo` (because that increases the code size without significant performance benefits). It also makes sense for `baz` to get inlined into `bar`. This is illustrated below [Ast13b]:



Packing the blocks is also done based on this call graph profiling. The most common cases will be put next to each other, and, where possible, subsequent steps are put next to each other. The more we can pack related code together, the fewer page faults we get by jumping to some other section, causing a cache miss... If the function being called is in the same page as the call, it has achieved "page locality" (and that is the goal!). This is represented visually [Ast13b]:



According to the author, the "dead" code goes in its own special block. I don't think they actually mean truly dead code, the kind that is compile-time determined to be unreachable, but instead they mean code that never gets invoked in any of the training runs.

So, to sum up, the training data is used to identify what branches are likely to be taken, inlines code where that is a performance increase, and tries to pack the binary code in such a way as to reduce cache misses/page faults. How well does it work?

Benchmark Results

This table, condensed from [Ast13b], summarizes the gains achieved. The application under test is a standard benchmark suite (Spec2K) (admittedly, C rather than Rust, but the goal is to see if the principle of POGO works and not just a specific implementation):

Spec2k:	**sjeng**	**gobmk**	**perl**	**povray**	**gcc**
App Size:	Small	Medium	Medium	Medium	Large
Inlined Edge Count	50%	53%	25%	79%	65%
Page Locality	97%	75%	85%	98%	80%
Speed Gain	8.5%	6.6%	14.9%	36.9%	7.9%

There are more details in the source as to how many functions are used in a typical run and how many things were inlined and so on. But we get enough of an idea from the last row of how much we are speeding up the program, plus some information about why. We can speculate about how well the results in a synthetic benchmark translate to real-world application performance, but at least from this view it does seem to be a net gain.

28 — Causal and Simulation Profiling

Causal Profiling

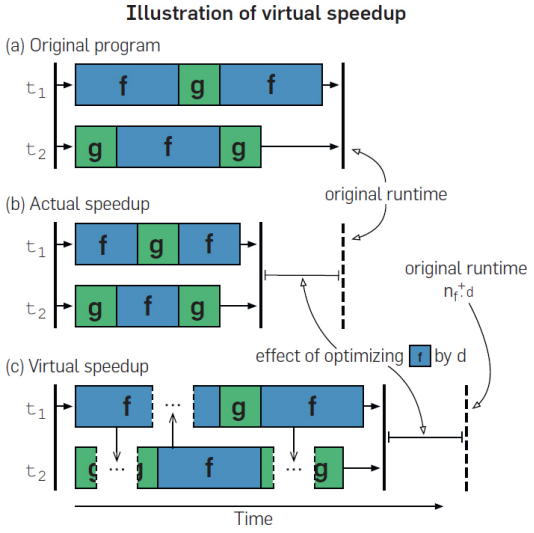
At this point we've got some experience in identifying areas of the program that we think are slow or are limiting the maximum performance of our program. If we are presented with more than one thing, how do we know which of those would yield the most benefit? Is it possible that optimizing something would actually have no effect or even a negative effect? The scientific approach would be to do an experiment and find out. That is, change the code, see the impact, re-evaluate. What causal profiling offers us is a way to run those experiments without changing any code. That could be a significant savings of time and effort.

One such causal profiler is called Coz (pronounced like "cause") [CB15]. It does a what-if analysis that says: what would be the impact of speeding up this particular part of the code?

A very simple approach would just look at the time of the program and just calculate what happens if the runtime of function `work()` is reduced by, say, 10%. But that isn't a realistic approach, because speeding up that function might have no overall impact or change the execution time by increasing lock contention or some other mechanism. No, we actually need a simulation.

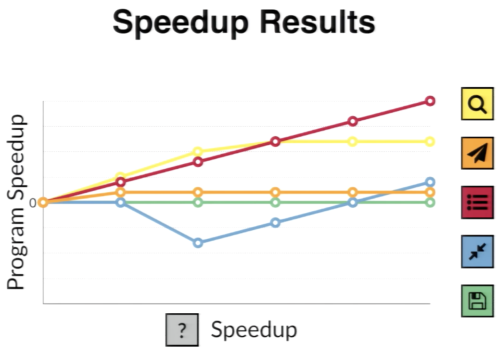
The key observation of the Coz profiler authors is the idea that speeding up some area of code is fundamentally the same as slowing down every other part of the code. It's all relative! This is what we would call a virtual speedup. How is the other code slowed down? Adding some pauses to (stopping the execution of) the other threads. That certainly makes them run slower. Maybe you're not convinced that slowing down everything else is equivalent?

Shown below is the argument from [CB15] in visual form. The original runtime is shown as (a). Hypothetically, say we make function `f` faster by some factor $d = 40\%$ that gives us (b). But, if instead of actually making function `f` faster, let's pause other threads for $0.4 \times t(f)$ where $t(f)$ is the original execution time of `f`. Then we get an equivalent effect to that of optimizing `f` by d.



The tool provides some graphs of potential program speedup. From the recorded presentation, it might look

something like the diagram below, where the boxes on the right correspond to different theoretical actions in the application being examined:



This pretend application shows all the possible outcomes, from continuous linear speedup, to speedup capped at some point, to no effect, and finally to where optimizing something makes the overall program runtime worse.

It's easy to imagine scenarios corresponding to each of those. If we're computing the n-body problem, anything that improves the calculation of forces will certainly make things better. And it's easy to imagine that sometimes optimizing a part of the program does not improve anything because that code is not on the critical path. We can also easily imagine that improving something works up to a point where it ceases to be the limiting factor. But making things worse? We already covered that idea: speeding up a thread may increase lock contention or add something to the critical path. At some point, things may recover and be a net benefit.

It is important to remember that just because hypothetically speeding up a particular part of the program would be beneficial, doesn't mean that it's possible to speed up that part. And almost certainly not possible to do so to an arbitrary degree. We still have to make an assessment of what optimizations we can make and how difficult it would be to actually realize those improvements.

Once we've made a change to the program, then it's time to run an experiment again with the new baseline to see what else we can do.

The paper has a table summarizing the optimizations they applied to a few different programs, which seems to support the idea that the tool can be used effectively to get a meaningful speedup with relatively few lines of code [CB15]:

	Summary of optimization results		
Application	**Speedup**	**Diff size**	**Source lines**
Memcached	9.39% ± 0.95%	−6, +2	10,475
SQLite	25.60% ± 1.00%	−7, +7	92,635
blackscholes	2.56% ± 0.41%	−61, +4	342
dedup	8.95% ± 0.27%	−3, +3	2,570
ferret	21.27% ± 0.17%	−4, +4	5,937
fluidanimate	37.5% ± 0.56%	−1, +0	1,015
streamcluster	68.4% ± 1.12%	−1, +0	1,779
swaptions	15.8% ± 1.10%	−10, +16	970

There are potentially some limitations to this, of course. Putting pauses in the execution of the code can work when the execution of the program is all on the same machine and we have control over all the threads; it would need some meaningful extension to work for a distributed system where coordinating things across many servers would be needed.

Using some benchmarking workload, the authors estimate a 17.6% overhead for this tool, which is broken down into 2.6% for startup debug information collection, sampling at 4.8%, and 10.2% is the delay that's caused by slowing down other threads to create a virtual speedup [CB15].

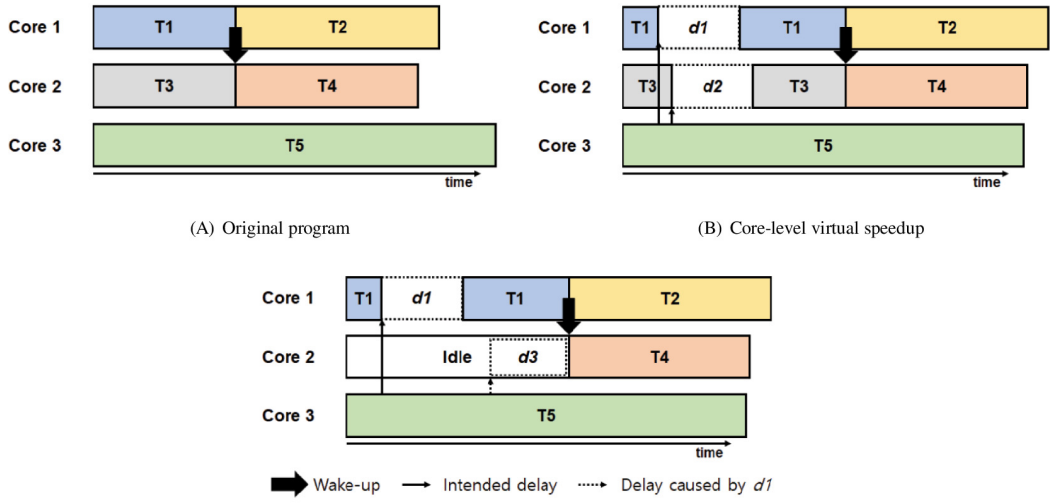
If you'd like to see some more details of the tool, you can see the author presentation at the ACM symposium here: `https://www.youtube.com/watch?v=jE0V-p1odPg`

The Next Generation: SCOZ

A few years later, another group of researchers worked to extend the Coz profiler to account for some limitations: inability to look at multi-process applications (which we just talked about), but also cases where the OS itself becomes a bottleneck [AKNJ21]. You may recall that looking into the kernel is not always permitted for a given profiling tool – depending on permissions. It is therefore obvious to imagine that if one cannot look, then one cannot touch.

If, indeed, the kernel is a bottleneck, it may or may not be possible to do very much about it. Avoiding system calls is one of the "do less work" strategies that you have likely already considered; oftentimes doing the system call is because we must. In an open-source OS or a kernel module we control, then we would have the ability to do something about it, if coz could give some recommendations.

The approach of [AKNJ21] is to move from a thread-based system to a core-based one. So where, previously, to simulate speedup of a particular section of code the tool would pause all other threads, now it pauses execution on all other cores. There are some details to consider about what to do when there are idle cores to make sure that the delay is charged to the "right" core when one is woken up by another. The diagram the authors provides helps to explain it well:



(A) Original program

(B) Core-level virtual speedup

(C) Core 2 is in idle state before thread 4 wakes up

The implementation involves one profiler thread for each core to manage its behaviour. The thread is pinned to the core, and its purpose is primarily to call the `ndelay` interface (with preemption temporarily disabled) to stop execution of code on that specific core [AKNJ21].

This extension does allow scoz to do more than coz, but it does require running within the kernel, which imposes certain limitations to where and how it can be applied. That's an interesting tradeoff!

0.1 Simulations

In a related idea,

Cachegrind

Cachegrind is another tool in the Valgrind package and this one is much more performance oriented than the other two tools. Yes, memcheck and Helgrind look for errors in your program that are likely to lead to slowdowns

(memory leaks) or make it easier to parallelize (spawn threads) without introducing errors. Cachegrind, however, does a simulation of how your program interacts with cache and evaluates how your program does on branch prediction. As we discussed earlier, cache misses and branch mispredicts have a huge impact on performance.

Recall that a miss from the fastest cache results in a small penalty (perhaps, 10 cycles); a miss that requires going to memory requires about 200 cycles. A mispredicted branch costs somewhere between 10-30 cycles. All figures & estimates from the cachegrind manual [Dev15].

Cachegrind reports data about the First Level Instruction Cache (I1) [L1 Instruction Cache], the First Level Data Cache (D1) [L1 Data Cache], and the Last Level Cache (LL) [L3 Cache].

Unlike the normal Valgrind operation, you probably want to turn optimizations on (compile the release version). You still want debugging symbols, of course, but enabling optimizations will tell you more about what is going to happen in the released version of your program.

If I instruct cachegrind to run on a simple ECE 252-type example using the `-branch-sim=yes` option (because by default it won't show it):

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search
==16559== Cachegrind, a cache and branch-prediction profiler
==16559== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==16559== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==16559== Command: ./search
==16559==
--16559-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16559==
==16559== I   refs:      310,670
==16559== I1  misses:      1,700
==16559== LLi misses:      1,292
==16559== I1  miss rate:    0.54%
==16559== LLi miss rate:    0.41%
==16559==
==16559== D   refs:      114,078  (77,789 rd   + 36,289 wr)
==16559== D1  misses:      4,398  ( 3,360 rd   +  1,038 wr)
==16559== LLd misses:      3,252  ( 2,337 rd   +    915 wr)
==16559== D1  miss rate:     3.8% (   4.3%     +    2.8%  )
==16559== LLd miss rate:     2.8% (   3.0%     +    2.5%  )
==16559==
==16559== LL refs:         6,098  ( 5,060 rd   +  1,038 wr)
==16559== LL misses:       4,544  ( 3,629 rd   +    915 wr)
==16559== LL miss rate:      1.0% (   0.9%     +    2.5%  )
==16559==
==16559== Branches:       66,622  (65,097 cond +  1,525 ind)
==16559== Mispredicts:     7,202  ( 6,699 cond +    503 ind)
==16559== Mispred rate:     10.8% (  10.2%     +   32.9%   )
```

So we see a breakdown of the instruction accesses, data accesses, and how well the last level of cache (L3 here) does. Why did I say enable optimization? Well, here's the output of the search program if I compile with the `-O2` option. Yes, this was a C program, but the idea is the same.

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search
==16618== Cachegrind, a cache and branch-prediction profiler
==16618== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==16618== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==16618== Command: ./search
==16618==
--16618-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16618==
==16618== I   refs:      306,169
==16618== I1  misses:      1,652
==16618== LLi misses:      1,286
==16618== I1  miss rate:    0.53%
==16618== LLi miss rate:    0.42%
==16618==
==16618== D   refs:      112,015  (76,522 rd   + 35,493 wr)
==16618== D1  misses:      4,328  ( 3,353 rd   +    975 wr)
==16618== LLd misses:      3,201  ( 2,337 rd   +    864 wr)
```

```
==16618== D1  miss rate:     3.8% (   4.3%     +    2.7%  )
==16618== LLd miss rate:     2.8% (   3.0%     +    2.4%  )
==16618==
==16618== LL refs:         5,980  ( 5,005 rd   +    975 wr)
==16618== LL misses:       4,487  ( 3,623 rd   +    864 wr)
==16618== LL miss rate:      1.0% (   0.9%     +    2.4%  )
==16618==
==16618== Branches:       65,827  (64,352 cond +  1,475 ind)
==16618== Mispredicts:     7,109  ( 6,596 cond +    513 ind)
==16618== Mispred rate:     10.7% (  10.2%     +   34.7%   )
```

Interesting results: our data and instruction miss rates went down marginally but the branch mispredict rates went up! Well sort of – there were fewer branches and thus fewer we got wrong as well as fewer we got right. So the total cycles lost to mispredicts went down. Is this an overall win for the code? Yes.

In some cases it's not so clear cut, and we could do a small calculation. If we just take a look at the LL misses (4 544 vs 4 487) and assume they take 200 cycles, and the branch miss penalty is 200 cycles, it went from 908 800 wasted cycles to 897 400; a decrease of 11 400 cycles. Repeat for each of the measures and sum them up to determine if things got better overall and by how much.

Cachegrind also produces a more detailed output file, titled `cachegrind.out.<pid>` (the PID in the example is 16618). This file is not especially human-readable, but we can ask the associated tool `cg_annotate` to break it down for us, and if we have the source code available, so much the better, because it will give you line by line information. That's way too much to show even in the notes, so it's the sort of thing I can show in class (or you can create for yourself) but here's a small excerpt from the `search.c` example:

```
--------------------------------------------------------------------------------
-- Auto-annotated source: /home/jz/ece254/search.c
--------------------------------------------------------------------------------
 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
127    1    1 96    3    0  4    0    0 23  11  0   0      for ( int i = arg->startIndex; i < arg->endIndex; ++i
    ) {
147    0    0 84    3    2  0    0    0 21   9  0   0          if ( array[i] == arg->searchValue ) {
  6    0    0  4    0    0  2    0    0  0   0  0   0              *result = i;
  2    0    0  0    0    0  0    0    0  0   0  0   0              break;
  .    .    .  .    .    .  .    .    .  .   .  .   .          }
  .    .    .  .    .    .  .    .    .  .   .  .   .      }
```

Cachegrind is very... verbose... and it can be very hard to come up with useful changes based on what you see... assuming your eyes don't glaze over when you see the numbers. Probably the biggest performance impact is last level cache misses (those appear as DLmr or DLmw). They have the highest penalty. You might also try to look at the Bcm and Bim (branch mispredictions) to see if you can give some better hints about what the likelihood of branch prediction is [Dev15].

29 — Liar, Liar

Here's a video of a helicopter flying: `https://www.youtube.com/watch?v=jQDjJRYmeWg`

The video's not fake; it's a real helicopter and it's really flying. What's happening is that the camera is taking images at some multiple of the frequency of the blade rotation speed. When playing back the sequence of images in the form of the video, it gives the illusion that the blades are not spinning at all. The illusion is not malicious or even intentional – nobody's trying to trick you – and yet, we are seeing the "wrong" thing.

> *In the criminal justice system, the people are represented by two separate, yet equally important, groups: the police, who investigate crime; and the district attorneys, who prosecute the offenders.*

If you have ever watched the TV series Law & Order, you will recognize that sentence is read over a title card before the cold open of the show. As part of their investigation, the police collect various pieces of evidence such as testimony, DNA, fingerprints, video. Then, in court, the prosecution and defence lawyers present their side, using the evidence to support their argument. Obviously, each side is presenting a different conclusion, and it's up to the judge or jury to decide which narrative they believe.

The actual criminal justice system is much more complex than that and there are rules of evidence and procedures and a beyond-a-reasonable-doubt standard of proof, to say nothing of competing narratives and the effects of how well the case was presented by the lawyer. Still, the analogy works well enough even if the profiler is a single program rather than two state agencies: there is the process of collecting evidence and then there is using the evidence to put together a narrative about what happened.

TV shows make it all very neat, because the story has to be told within the time constraints of the episode or season. Something like saying the suspect's DNA is found at the crime scene is treated the definitive proof that they did the murder and it's off to prison for them. DNA is an excellent tool for evidence purposes and it's been critical in convicting criminals and freeing the wrongly-convicted. But there's nuance on DNA evidence that makes it much less definitive than a police procedural TV show makes it seem.

In reality, a DNA expert would testify that there is a match between the DNA collected at the scene and the DNA sample collected from the suspect, and that match is reported with a certain percentage of certainty (or alternatively, phrased as something like "there is a 1 in X chance that the crime scene DNA came from someone other than the suspect"). Those rare things *do* happen, of course, though with astronomically small odds, it's easy to reach the threshold of beyond-a-reasonable-doubt. Even so, that's just the evidence – the presence of the DNA only proves that the suspect was at the scene; there could be another reason why the suspect was there (before, during, or after the crime). Thus, the presence of the suspect's DNA is *consistent* with the prosecution's theory of the crime, but does not conclusively prove anything on its own. Only the totality of the evidence, considered together, would lead to a conclusion about whether that narrative is true.

We'll consider some examples where the profiler gets it wrong on each part: either the data collection or the narrative that we get is incorrect. Going back to the helicopter example: the samples themselves are perfectly fine, but the periodic sampling strategy has a weakness. The measured positions of the helicopter blades are all correct, and yet the narrative that this builds is that the blades don't move. We know that's wrong because of our human judgement, and you need to apply that same judgement to what a tool is telling you.

Sampling is something we know profiling tools do, and profilers are useful tools, but they can mislead you. If you understand how profilers work – and some common pitfalls – you won't draw the wrong conclusions. Sampling-based profiler can miss things, while an instrumentation-based profiler distorts the system under observation. The

main assumptions underlying sampling are that samples are "random" and that the sample distribution approximates the actual time-spent distribution[63].

Lies from Metrics

The helicopter video example introduced the first type of lie that we might get from metrics: what are we *not* seeing? Periodic sampling misses any events that take place only between the samples. So if a particular function is a periodic interrupt handler code and the frequency of that just so happens to make it run between samples, as far as your profiler is concerned that code didn't run at all – even though we could observe from external signs (e.g., the interrupt actually being handled!) that this is not correct.

Let's talk about CPU perf counters. Remember that these are things that the CPU tracks automatically at all times – no need to change the program or interrupt it to get this data. In particular, we're going to look at two types of sampling-based lies. The reference for the first type of lie is a blog post by Paul Khuong [Khu14].

This goes back to `mfence`, which we've seen before. It is used, for instance, in spinlock implementations. Khuong found that his profiles said that spinlocking didn't take much time. But empirically: eliminating spinlocks produced better results than expected! Why?

The next step is to create microbenchmarks to better understand what's going on. The microbenchmark contained memory accesses to uncached locations, or computations, surrounded by store pairs/`mfence`/locks. He used perf to evaluate the impact of `mfence` vs lock. You'll recall that perf is sampling-based and records how often the CPU is found executing each instruction. The leftmost column in the table below shows the percentage of time that instruction took. They don't sum to 100 because of rounding and also other instructions not shown to keep the size of the example reasonable.

```
# for locks:
$ perf annotate -s cache_misses
[...]
    0.06 :        4006b0:       and    %rdx,%r10
    0.00 :        4006b3:       add    $0x1,%r9
    ;; random (out of last level cache) read
    0.00 :        4006b7:       mov    (%rsi,%r10,8),%rbp
   30.37 :        4006bb:       mov    %rcx,%r10
    ;; foo is cached, to simulate our internal lock
    0.12 :        4006be:       mov    %r9,0x200fbb(%rip)
    0.00 :        4006c5:       shl    $0x17,%r10
    [... Skipping arithmetic with < 1% weight in the profile]
    ;; locked increment of an in-cache "lock" byte
    1.00 :        4006e7:       lock incb 0x200d92(%rip)
   21.57 :        4006ee:       add    $0x1,%rax
    [...]
    ;; random out of cache read
    0.00 :        400704:       xor    (%rsi,%r10,8),%rbp
   21.99 :        400708:       xor    %r9,%r8
    [...]
    ;; locked in-cache decrement
    0.00 :        400729:       lock decb 0x200d50(%rip)
   18.61 :        400730:       add    $0x1,%rax
    [...]
    0.92 :        400755:       jne    4006b0 <cache_misses+0x30>
```

In the lock situation, reads take $30 + 22 = 52\%$ of runtime, while locks take $19 + 21 = 40\%$ of runtime.

```
# for mfence:
$ perf annotate -s cache_misses
[...]
    0.00 :        4006b0:       and    %rdx,%r10
    0.00 :        4006b3:       add    $0x1,%r9
    ;; random read
    0.00 :        4006b7:       mov    (%rsi,%r10,8),%rbp
   42.04 :        4006bb:       mov    %rcx,%r10
    ;; store to cached memory (lock word)
    0.00 :        4006be:       mov    %r9,0x200fbb(%rip)
    [...]
```

[63] Lifted from "Profilers are Lying Hobbitses", `https://www.infoq.com/presentations/profilers-hotspots-bottlenecks/`, which talks about profiling for JVMs.

```
 0.20 :        4006e7:       mfence
 5.26 :        4006ea:       add    $0x1,%rax
 [...]
 ;; random read
 0.19 :        400700:       xor    (%rsi,%r10,8),%rbp
43.13 :        400704:       xor    %r9,%r8
 [...]
 0.00 :        400725:       mfence
 4.96 :        400728:       add    $0x1,%rax
 0.92 :        40072c:       add    $0x1,%rax
 [...]
 0.36 :        40074d:       jne    4006b0 <cache_misses+0x30>
```

Looks like the reads take 85% of runtime, while the `mfence` takes 15% of runtime.

That makes it seem like the `mfence` approach is clearly better: a significantly lower percentage of time is going towards the overhead of concurrency control. But empirically, the observed behaviour says this is worse. Why the mismatch?

There are two things going on here. The first is that while the *percentage* of time going to the concurrency control might be lower (more on this in a second), the total execution time is much higher. Compare the total # of cycles:

No atomic/fence:	2.81e9 cycles
lock inc/dec:	3.66e9 cycles
`mfence`:	19.60e9 cycles

That 15% number is a total lie. Profilers, even using CPU counts, drastically underestimate the impact of `mfence`, and overestimate the impact of locks. This is because `mfence` causes a pipeline flush, and the resulting costs get attributed to instructions being flushed, not to the `mfence` itself. In other words, `mfence` makes other instructions run more slowly, which camouflages its own effect on the overall performance.

Even if `mfence` actually were better by percentage, it's not a win to have a lower percentage of a larger total time.

Skid. Another cause for the attributions of which instructions are expensive being wrong is something called *skid*. The perf wiki describes it as follows: "Interrupt-based sampling introduces skids on modern processors. That means that the instruction pointer stored in each sample designates the place where the program was interrupted to process the PMU interrupt, not the place where the counter actually overflows..."; the counter overflowing is the trigger to take the next sample.

A made-up example of what that might look like:

```
ld r1,0x12341234     0.1%
add r2,r3            1.0%
sub r3,r4            1.0%
NOP                 27.0%
```

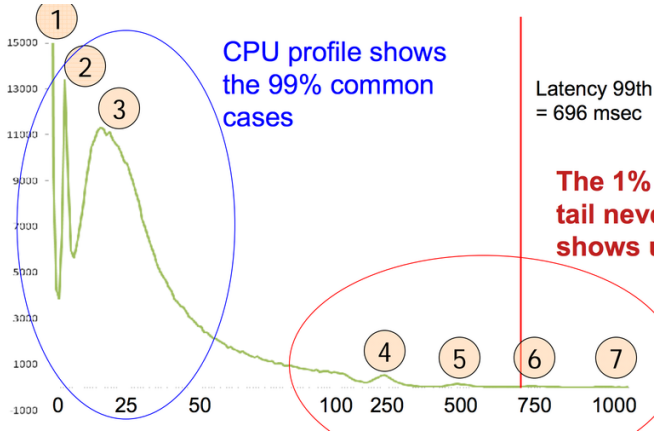
The NOP is obviously not the cause here; it's the load instruction that really is the expensive one.

Modern Intel and AMD x86_64 CPUs introduce the ability to have more precise (low- or no-skid) sampling take place with some hardware support. Why would we not use this all the time? It's only a guess, but there's a possibility it slows down execution, can't be used all the time, or might give even more distorted results in particular scenarios.

The Long Tail

The other type of lie that sampling can hide is the one where infrequent long tails are hidden in averages. Our source here is the blog post by Dan Luu [Luu16]. Suppose we have a task that's going to get distributed over multiple computers (like a search). If we look at the latency distribution, the problem is mostly that we see a long tail of events and when we are doing a computation or search where we need all the results, we can only go as

the slowest step. Let's take a look at a histogram of disk read latencies, where we are performing a 64 kB read, also from that source[64]:

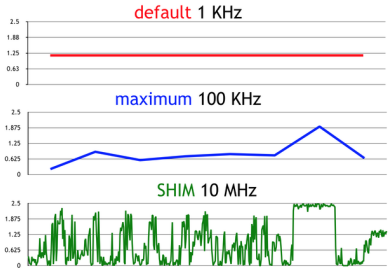


Let's break it down. Peak 1 corresponds to something cached in RAM—best case scenario. Peak 2 is at around 3ms, which is too fast for spinning and seeking magnetic hard disks, but it's fast enough for reading something from the disk cache via the PCI-Express interface. Peak 3 is obviously disk seek and read times, around 25ms.

These numbers don't look terrible, except for the fact that we have peaks at 250, 500, 750, and 1000 ms and the 99th percentile is some 696ms which is a very, very long time. Sampling profilers are not very good at finding these things, because they throw everything into various buckets and therefore we get averages. The averages are misleading, though, because we have these weird outliers that take dramatically longer. Averages are nice as long as our data is also reasonably "nice".

So what actually happened? Well, from [Luu16]: The investigator found out that the cause was kernel throttling of the CPU for processes that went beyond their usage quota. To enforce the quota, the kernel puts all of the relevant threads to sleep until the next multiple of a quarter second. When the quarter-second hand of the clock rolls around, it wakes up all the threads, and if those threads are still using too much CPU, the threads get put back to sleep for another quarter second. The phase change out of this mode happens when, by happenstance, there aren't too many requests in a quarter second interval and the kernel stops throttling the threads. After finding the cause, an engineer found that this was happening on 25% of disk servers at Google, for an average of half an hour a day, with periods of high latency as long as 23 hours. This had been happening for three years.

Further limitations of sampling profilers emerge, as demonstrated in this graph, also from [Luu16], showing the data we get out of our sampling profiler if we take a look at Lucene (a search indexer):



So at the default sampling interval for perf we see…nothing interesting whatsoever. If we bump up to the max sampling frequency of perf, we get a moderately more interesting graph, but not much. If we use a different tool and can sample at a dramatically higher rate, then we end up with something way more useful. So we're left to wonder why does perf sample so infrequently, and how does SHIM get around this?

Well, for one thing, perf samples are done with interrupts. Processing interrupts takes a fair amount of time and if you crank up the rate of interrupts, before long, you are spending all your time handling the interrupts rather than

[64]The image is cropped in the original source too – I've not been able to find one that shows the full text correctly

doing useful work. So sampling tools usually don't interrupt the program too often. SHIM gets around this by being more invasive—it instruments the program, adding some periodically executed code that puts information out whenever there is an appropriate event (e.g., function return). This produces a bunch of data which can be dealt with later to produce something useful.

This instrumentation-based approach is more expensive in general, but note that DTrace[65] and Nethercote's counts tool (discussed in L27) also enable custom instrumentation of select events.

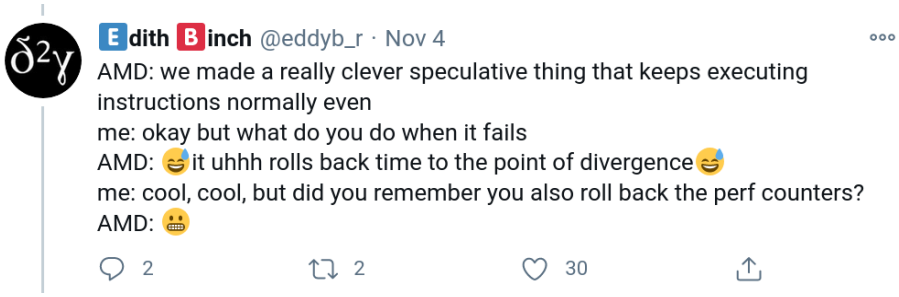
Lies from Counters

This is fairly niche, but Rust compiler hackers were trying to include support for hardware performance counters (what perf reports) because `-Z self-profile` data was too noisy[66]. Counters are, for instance, faster than measuring time and way (i.e. 5 orders of magnitude) more deterministic.

To make counters as deterministic as possible:

- disable Address Space Layout Randomization (security mitigation; but, randomized pointer addresses affect hash layouts);
- subtract time spent processing interrupts (IRQs);
- profile one thread only (if you can, in your context).

Fun fact. We talked about Spectre back in Lecture 7. Speculative execution comes up here too in terms of counters being wrong. AMD speculates past atomics and then rolls back, but doesn't roll back perf counters. Post-Spectre, there's a hidden model-specific register ("SpecLockMap") that disables speculating past atomics, the kind of thing you would want around to protect you against future things in that vein that someone might discover. Or, in better words than mine[67]:



Lies about Calling Context

This part is somewhat outdated now, as it's a pretty specific technical problem that especially arises under the `gprof` tool. It's still a good example of lying tools, though, so we'll cover a condensed version. Yossi Kreinin [Kre13] writes about it in more detail.

`gprof` uses two C standard-library functions:

- **profil()**: asks glibc to record which instruction is currently executing (100×/second).
- **mcount()**: records call graph edges; called by `-pg` instrumentation.

Hence, **profil** information is statistical, while **mcount** information is exact. `gprof` can draw unreliable inferences by combining these pieces of data. That happens in other domains too – apportionment of fault in the law sometimes has the problem where the judge can't figure out whose story is more likely and they just decide to assign

[65] Note also the comment in the blog post: "Yes, that includes dtrace, which I'm calling out in particular because any time you have one of these discussions, a dtrace troll will come along to say that dtrace has supported that for years. It's like the common lisp of trace tools, in terms of community trolling."

[66] Full story, in gory detail, at `https://hackmd.io/sH315lO2RuicY-SEt7ynGA?view`.

[67] `https://twitter.com/eddyb_r/status/1323587371703668742`

fault 50-50, even if that's not sensible. The profiler doesn't have to choose between competing narratives, but it does have to come to a conclusion based on incomplete information and sometimes that conclusion is wrong.

Suppose you have a method `easy` and a method `hard`, each of which is called once, and `hard` takes up almost all the CPU time. `gprof` might divide total time by 2 and report bogus results where it says both methods take an equal amount of time.

Some examples of `gprof` results that are suspect are contribution of children to parents and the total runtime spent in self+children. Call graph edges are correct when considering functions with only one caller (e.g. `f()` only called by `g()`) or functions which always take the same time to complete (e.g. `rand()`). On the other hand, results for any function whose running time depends on its inputs, and is called from multiple contexts, are sketchy.

Overall summary. We saw a bunch of lies that our tools can return: calling-context lies and perf attribution lies. To avoid being bitten by lies, remember to focus on the metric you actually care about, and understand how your tools work. If a result doesn't make sense, dig into it and validate if that really is the case.

30 — Clusters & Cloud Computing

Clusters and Cloud Computing

Almost everything we've seen so far has improved performance on a single computer. Sometimes, you need more performance than you can get on a single computer, or you have reached a point where it's a lot better value for your money to buy two cheap computers than one very expensive computer. For the most part, if you could split your problem into multiple threads or multiple processes, you can do the same with multiple computers. We'll survey techniques for programming for performance using multiple computers; although there's overlap with distributed systems, we're looking more at calculations here rather than coordination mechanisms.

Message Passing. Rust encourages message-passing, but a lot of your previous experience when working with C may have centred around shared memory systems and you may have done that in Rust too (`Mutex`, anyone?). In some circumstances that we've seen, you don't have a choice! The model of GPU programming meant we had to explicitly manage copying of data. When we have multiple computers, shared memory is not an option, so we have to use message passing. Fortunately, we know how to do that. The only thing to note now is that communication over the network is much more expensive than communicating within the same system, so we will need to think carefully about how much to communicate.

There was a time when we would talk about MPI, the *Message Passing Interface*, a de facto standard for programming message-passing multi- computer systems. This is, unfortunately, no longer the way. MPI sounds good, but in practice people tend to use other things. Here's a detailed piece about the relevance of MPI as of 10 years ago: [Dur15], if you are curious.

REST We've already seen asynchronous I/O using HTTP (curl) which we could use to interact with a REST API as one mechanism for multi-computer communication. You may have also learned about sockets and know how to use those, which would underlie a lot of the mechanisms we're discussing. The socket approach is too low-level for what we want to discuss, while the REST API approach is at a reasonable level of abstraction.

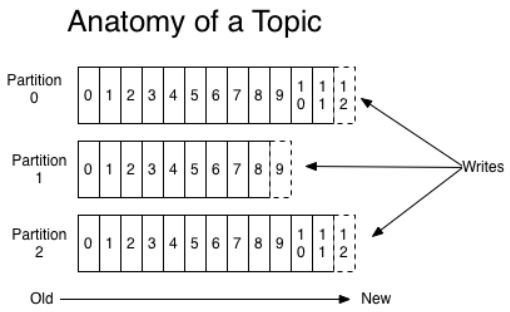
REST APIs are often completely synchronous, but don't have to be: you can set up callbacks to be notified when a computation is done, or the caller can check back later to see if the request is finished. But these aren't truly asynchronous, because the remote machine has to be available at the time of each call. Perhaps we'd like to decouple services even more than that and use something else...

Kafka. Let's talk about Apache Kafka, a self-described "distributed streaming platform" that's got significant adoption in industry as a mechanism of communication between different services running over a network. We're using [Kur20] as a guide and it provides a breakdown on how Kafka works.

Communication is based around the idea of producers writing a record (some data element, like an invoice) into a topic (categorizing messages) and consumers taking the item from the topic and doing something useful with it. A message remains available for a fixed period of time and can be replayed if needed. I think at this point you have enough familiarity with the concept of the producer-consumer problem and channels/topics/subscriptions that we don't need to spend a lot of time on it.

Kafka's basic strategy is to write things into an immutable log. The log is split into different partitions; you choose how many when creating the topic, where more partitions equals higher parallelism. The producer writes something and it goes into one of the partitions. Consumers read from each one of the partitions and writes down

its progress ("commits its offset") to keep track of how much of the topic it has consumed. See this image from `kafka.apache.org`:



The nice part about such an architecture is that we can provision the parallelism that we want, and the logic for the broker (the system between the producer and the consumer, that is, Kafka) is simple. Also, consumers can take items and deal with them at their own speed and there's no need for consumers to coordinate; they manage their own offsets. Messages are removed from the topic based on their expiry, so it's not important for consumers to get them out of the queue as quickly as possible.

You can see a visualization of Kafka at https://softwaremill.com/kafka-visualisation/.

Quick aside on that: under normal circumstances, in something like a standard queue, there's a little bit of pressure to get items out of the queue quickly. If the queue is growing, it might mean the queue is full and new items can't be produced or are thrown away... or perhaps you get charged money for the storage space the queue is using.

You might think that it's a solution to take the item out of the queue in one transaction and then process it later. That's okay only if you've successfully saved it to a database or other persistent storage. Otherwise, you could take the item out and a crash or called shutdown means that the item doesn't actually get processed. Oops!

Alternatives. There's also some popular AWS (Amazon Web Services)-based solutions for this kind of multiple computer communication: SNS (Simple Notification Service) and SQS (Simple Queueing Service). They are, broadly speaking, just other ways to decouple the communication of your programs.

SNS is good for sending lots of messages to multiple receivers, maybe push notifications or making sure all systems update their records or that you send pager alerts to people about things that have gone wrong. SNS messages are not persistent, so if you miss it, you miss it.

SQS is more for batches of work where it's not particularly time-sensitive and the item will be consumed by a worker. SQS data is deleted after being taken out of the queue and may or may not have ordering guarantees. SQS does have limits on how long a message can be stored, though.

Cloud Computing

We'll start with a little bit of history. In the old days, if you wanted a cluster, you had to find a bunch of money to buy and maintain a pile of expensive machines. Not anymore. Cloud computing is perhaps way overhyped, but we can talk about one particular aspect of it, as exemplified by Amazon's Elastic Compute Cloud (EC2).

Consider the following evolution:

- Once upon a time, if you wanted a dedicated server on the Internet, you had to get a physical machine hosted, usually in a rack somewhere. Or you could live with inferior shared hosting.
- Virtualization meant that you could instead pay for part of a machine on that rack, e.g. as provided by `slicehost.com`. This is a win because you're usually not maxing out a computer, and you'd be perfectly happy to share it with others, as long as there are good security guarantees. All of the users can get root access to their virtual part computer. (Unfortunately, Spectre and Meltdown allow you to see parts of the machine that are not yours.)

- Clouds enable you to add more machines on-demand. Instead of having just one virtual server, you can spin up dozens (or thousands) of server images when you need more compute capacity. These servers typically share persistent storage, also in the cloud.

In cloud computing, you pay according to the number of machines, or instances, that you've started up. Providers offer different instance sizes, where the sizes vary according to the number of cores, local storage, and memory. Some instances even have GPUs, but it seemed uneconomic to use this for Assignment 3. Instead we have the `ecetesla` machines.

Launching Instances. When you need more compute power, you launch an instance. The input is a virtual machine image. You use a command-line or web-based tool to launch the instance. After you've launched the instance, it gets an IP address and is network-accessible. You have full root access to that instance.

Amazon provides public images which run a variety of operating systems, including different Linux distributions, Windows Server, and OpenSolaris. You can build an image which contains the software you want, including Hadoop and OpenMPI.

Terminating Instances. A key part of cloud computing is that, once you no longer need an instance, you can just shut it down and stop paying for it. All of the data on that instance goes away.

Continuous Deployment. The combination of launching and terminating instances can mean that updates don't require downtime: you start up the new instances and then shut down the old ones so there's no gap where the service is down.

Sometimes this causes slight headaches, like if you make a database change that's incompatible: the old nodes will have a problem, so you need to be a little careful with this.

Storing Data. You probably want to keep some persistent results from your instances. Basically, you can either mount a storage device, also on the cloud (e.g. Amazon Elastic Block Storage); or, you can connect to a database on a persistent server (e.g. Amazon SimpleDB or Relational Database Service); or, you can store files on the Web (e.g. Amazon S3).

Clusters versus Laptops

There is a paper about this: Frank McSherry, Michael Isard, Derek G. Murray. "Scalability! But at what COST?" HotOS XV. This part of the lecture is based on the companion blog post [McS15].

The key idea: scaling to big data systems introduces substantial overhead. Let's just see how, say, a laptop compares, in absolute times, to 128-core big data systems.

Summary. Big data systems haven't yet been shown to be obviously good; current evaluation is lacking. The important metric is not just scalability; absolute performance matters a lot too. We don't want a situation where we are just scaling up to n systems to deal with the complexity of scaling up to n systems. Or, as Oscar Wilde put it: "The bureaucracy is expanding to meet the needs of the expanding bureaucracy."

Methodology. We'll compare a competent single-threaded implementation to top big data systems, as described in an OSDI 2014 (top OS conference) paper on GraphX [GXD+14]. The domain: graph processing algorithms, namely PageRank and graph connectivity (for which the bottleneck is label propagation). The subjects: graphs with billions of edges, amounting to a few GB of data.

Results. 128 cores don't consistently beat a laptop at PageRank: e.g. 249–857s on the twitter_rv dataset for the big data system vs 300s for the laptop, and they are 2× slower for label propagation, at 251–1784s for the big data system vs 153s on twitter_rv. From the blogpost:

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread	1	300s	651s

Label propagation to fixed-point (graph connectivity)

System	cores	twitter_rv	uk_2007_05
Spark	128	1784s	8000s+
Giraph	128	200s	8000s+
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread	1	153s	417s

Wait, there's more. I keep on saying that we can improve algorithms for additional performance boosts too. But that doesn't generalize, so it's hard to teach. In this case, two improvements are: using Hilbert curves for data layout, improving memory locality, which helps a lot for PageRank; and using a union-find algorithm (which is also parallelizable). "10× faster, 100× less embarrassing". We observe an overall $2\times$ speedup for PageRank and $10\times$ speedup for label propagation.

Takeaways. Some thoughts to keep in mind, from the authors:

- "If you are going to use a big data system for yourself, see if it is faster than your laptop."
- "If you are going to build a big data system for others, see that it is faster than my laptop."

Movie Hour

Let's take a humorous look at cloud computing: James Mickens' session from Monitorama PDX 2014.

`https://vimeo.com/95066828`

31 — Introduction to Queueing Theory

A Short Introduction to Queueing Theory

Queueing theory is literally the theory of queues—what makes queues appear, how will they behave, and how do we make them go away? Queueing theory has played a role in your life whether you know it or not: this is how tech support at Rogers or Bell or Telus or whomever decides just how many customer service agents to have available at any given time. Of course, your local telecom chooses to minimize the number of employees at the cost of making you wait ("Your call is important to us; please hold while we ignore it.") but they study carefully how much waiting is too much waiting and how much is too little. Queueing theory is applicable to lots of fields, including industrial design, call centres, telecom systems, and computers executing transactions.

To scale up a system, we have a lot of choices to make, and these will work best if they are supported by data. Queueing theory helps us decide what's best. Here are a few possible examples, from [HB13]:

- Given a choice between a single machine with speed s or n machines, each with speed s/n, which should we choose?
- If the arrival rate and service rate double, how does the mean response time change?
- Should we try to balance load or is that a waste of time/effort?
- Can we give priority to certain operations without harming another category of job?
- How do job size variability and heavy-tailed workloads affect our choices of scheduling policy?
- If 12 servers is enough to handle 9 jobs per second, do we need 12 000 servers if we have an arrival rate if 9 000 jobs per second?

I tend to tell stories about banks that imply I hate them. Not really, they're just a place where there's likely to be a queue and I'm likely to be annoyed and thinking about how to optimize this situation. So let's define some terms formally, to make sure we're all on the same page when it comes to terminology and language. Some of these will seem obvious, but let's be complete (like the book [Liu09]):

- Server—The banking centre fulfilling customer requests.
- Customer—The initiator of service requests.
- Wait time—The time a customer spends waiting in line.
- Service time—The time from when a teller starts to serve a customer up to the time when the next customer is called forward.
- Arrival rate—The rate at which customers arrive.
- Service rate—The rate at which customer requests are serviced.
- Utilization—The fraction of the teller's time used actually handling customer requests (not idling).
- Queue length—The total number of customers waiting, or currently with a teller, or both.

- Response time—The sum of wait and service time for a single visit.
- Residence time—The total response time if a customer visits several tellers (or the same one multiple times).
- Throughput—The rate at which customers get their requests serviced and dealt with.

The mathematical symbols for this are represented in the following table [Liu09]:

Symbol	Semantics
S	Service time
V	Number of visits to the server
D	Service demand
R	Response time
R'	Residence time
X	Throughput
λ	Arrival rate
U	Utilization
W	Wait time
N	Total queue length (waiting and/or being serviced)

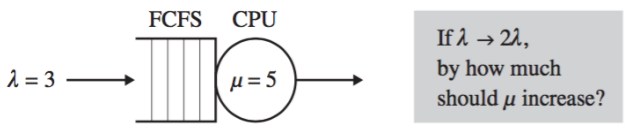
If you're cynical like me, you will think the bank works very hard to not have tellers in the bank to maximize your wait time and minimize their staffing costs. It's actually (allegedly) a trade-off: if I have to wait too long to do my banking, I could always take my business elsewhere (but is that likely to happen?). Minimizing customer wait time makes customers happy, so that's something the bank should want. It would also be nice if the bank trains its tellers well, so they can complete all operations, even unusual ones, quickly and efficiently, reducing the service time. The bank is not a charity operation so they will of course want to minimize staffing, but it knows that overstaffed is bad and understaffed is also bad.

Back to the realm of computers: you have lots of queues in your computer. The CPU uses a time-sharing scheduler to run as many concurrent programs as possible. A router has a queue for packets (data) that has a maximum size, and if this is exceeded, packets will be simply dropped.

Queueing theory gives us a formal framework with which to grapple with our problems instead of just guessing. Remember how bad we are at guessing.

Example. Let's look at a simple example from [HB13]. Imagine we have a system with one CPU that serves a queue of jobs in First-Come-First-Served (FCFS) order with an arrival rate λ of 3 jobs per second. Each job takes some amount of time and resources, but we can ignore the particulars for right now. Suppose the average service rate μ is 5 jobs per second (or stated another way, the average job requires $0.2s$ to service). The system is not overloaded: 3 jobs per second arriving is less than 5 jobs per second being serviced. Our terminology for describing the mean response time will be $E[T]$.

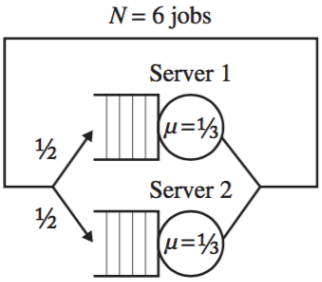
Suppose now that your boss says that tomorrow the arrival rate will double. If you do nothing, you can imagine, there will be a problem: we would have 6 jobs arriving per second, on average, to a system that can service, on average, 5 jobs per second. You have been allocated some budget to replace the CPU with a faster one, and you should choose one so that the jobs still have a mean response time of $E[T]$. This situation is depicted below [HB13]:



That is, customers should not notice the increase in arrival rate. So, should we (1) double the CPU speed; (2) more than double the CPU speed; or (3) less than double the CPU speed?

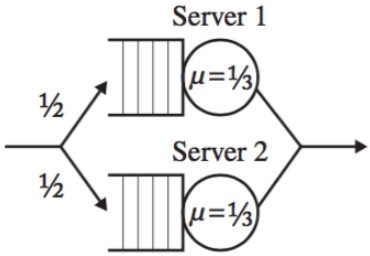
The answer is (3): we don't need to double the CPU speed. We can see later in formal terms why this is the case, but think for a minute about why it is? If we double the service rate and double the arrival rate, we actually get half the mean response time...

Example 2. Okay, how about another example from [HB13]. There are always $N = 6$ jobs running at a time. As soon as a job completes, a new one is started (this is called a *closed system*). Each job goes through to be processed on one of two servers (and it is 50-50 where the job ends up), each of which has a service time μ of 1 job per 3 seconds. Again, depicted below [HB13]:



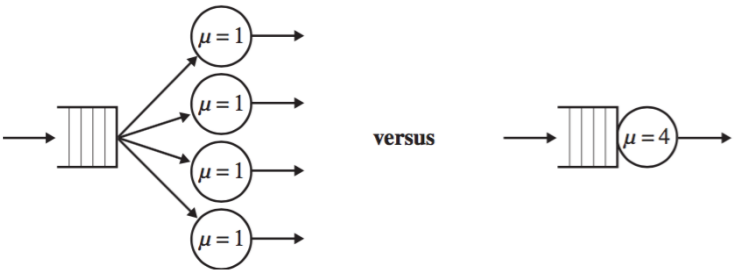
Bad news: sometimes, improvements do nothing. If we replace server 1 with one which is twice as fast (so 2 jobs per 3 seconds), does that help? Nope. Not really. Does raising N help? Nope, negligible effect. The bottleneck device is the limiting factor. Strangely, dropping N to 1 means the server replacement makes a difference, if you can call that improvement.

What if it's an *open system* where arrival times are independent of completion, as below [HB13]?



In this case, yes, replacing server 1 makes a huge difference!

Example 3. A third example, this time addressing directly the question of do we want one fast server or n slower ones? Horse-sized duck and duck-sized horses jokes aside, what is better if we want to minimize the mean response time when we have non-preemptable jobs (i.e., once started, a job has to run to completion and cannot be interrupted) [HB13]:



The answer is "it depends". That's frustrating, but this is Sparta. Or at least, real life. One big factor is the variability of the job sizes. Imagine you are at the grocery store and most people have 12 items or fewer[68] and

[68] Not less. Fewer. It is countable; therefore fewer. Yes, I am obsessive about this.

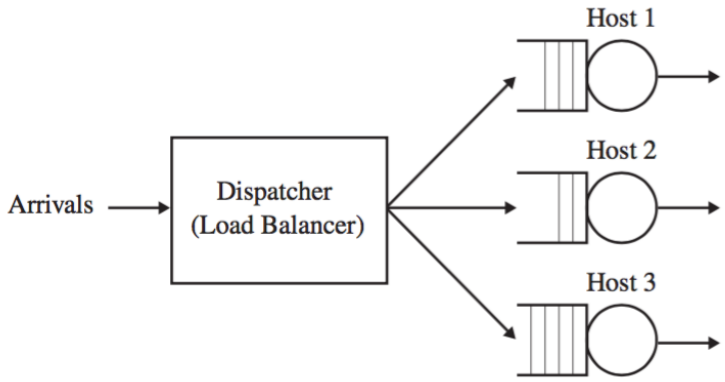
there's one guy who's buying 85 items. You don't want to be standing in line with milk and eggs behind someone who is trying to buy six of everything, do you? So if there's high variability, you probably want multiple servers—the guy buying the whole store can hold up line #1 and you can go to line #4 and you're out and done before he is finished.

What if the load is low? Chances are you would prefer the one fast server instead of having some number of servers doing nothing.

What if jobs are interruptible (preemptible)? You could always use a single fast machine to simulate n slow machines, so a single fast machine is at least as good as the alternative.

A Digression on Load Balancing

Imagine your typical "server farm"—you have n servers that are all responsible for handling incoming requests. Let's imagine all servers are the same (or close enough). What we typically see in load balancing is assignment of tasks to servers via some dispatcher [HB13]:



This isn't the only kind of load balancing we can do; there is also the ability to do after-the-fact assignment (or work-stealing), which consists of monitoring the various queues and reassigning work if it's piling up somewhere.

There are a few different task assignment policies—ways in which we can assign work to servers [HB13]:

- Random: Exactly what it sounds like.
- Round-Robin: The ith job goes to host i modulo n.
- Shortest-Queue: The job goes to the server with the shortest queue.
- Size-Interval-Task-Assignment: Short jobs go to one server, medium to another, long to another...
- Least-Work-Left: A job goes to the server that has the least total remaining work, where work is the sum of the size of the jobs.
- Central-Queue: Rather than being assigned to a host directly, when a server needs work to do, it gets the first job in the central queue.

Which of these policies yields the lowest mean response time? Answer: truthfully, nobody knows. It depends, of course, on your job variability and and that sort of thing, but it hasn't been well studied. PhD, anyone?

Red Line Overload[69]

Earlier I mentioned it would probably be bad to see 6 jobs arriving per second to a system that can handle 5 per second. This doesn't seem like rocket science, but it bears repeating. In our discussion we require that $\lambda \leq \mu$ and

[69] You've seen "Top Gun", right? `https://www.youtube.com/watch?v=siwpn14IE7E`

assume that $\lambda < \mu$. That is to say, we are not overloaded (as engineering students you may be amused by the idea that you might one day NOT be overloaded). Remember now that the values for λ and μ are averages, so it could happen that temporarily we "fall behind" a bit, but then make up for it a little later on, or we temporarily get ahead before a bunch more work gets piled on. Think about the long term, though—if we are not at least keeping up then this will eventually get out of hand. How badly? Well, in the limit, the queue length goes to infinity.

The justification comes from [HB13]: Let's represent time with t, its usual symbol, and define $N(t)$ as the number of jobs in the system at time t. $A(t)$ represents arrivals by time t and $D(t)$ represents departures by time t. So:

$$E[N(t)] = E[A(t)] - E[D(t)] \geq \lambda t - \mu t = t(\lambda - \mu)$$

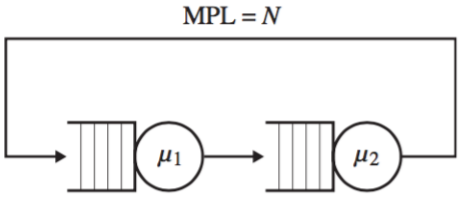
The tiniest bit of calculus says that if arrivals exceed departures, taking the limit as t goes to infinity means $t(\lambda - \mu)$ also goes to infinity. Whoops. So to prevent this terrible situation we just happily assume that this doesn't happen[70].

Raising μ is generally desirable. This is, after all, programming for performance – the faster we complete work, the more work we can get done in the same amount of time. Improving the service rate, however, does not necessarily improve the throughput.

Wait, what? We've assumed that the arrival rate is less than the service rate. So we have enough capacity to handle all incoming work. So the limiting factor on completed work is actually arriving work. We have the capacity to do at least what is arriving and possibly a bit more. Adding more work capacity doesn't mean more work gets done if there isn't any more work to do. You might be capable of completing six assignments for this class in a term, but if you're (mercifully) only assigned four, then you will only complete four. So raising μ increases the maximum possible throughput, but does not necessarily increase the actual throughput.

But just to make you suffer, things are very different in a closed system: one in which there is always more work to do and as soon as one item is finished the next one enters the queue. This is the case in batch systems. You know, the old mainframe kind of processing where you submit your job to be run overnight and in the morning you get a result. Hopefully the one you wanted. In that case, we are running at capacity all the time, so actually μ is the controlling factor – the throughput is exactly the service rate.

Not that open networks are particularly intuitive, but closed networks can kind of mess with our intuition in general. Imagine we have a closed system with Multiprogramming Level (MPL) of N as below [HB13]:



What is the throughput here? Intuition suggests $\min(\mu_1, \mu_2)$, right? Sometimes. This is okay if the slower server is always busy, but that's not always the case. What if N is 1? Okay, that's a bit of an exception case though. What about N being 2? Then the slower server has some work to do at all times right? Nope, sadly not. Sometimes the slow server is faster than the fast server, because μ_1 and μ_2 are just averages. And averages can be misleading! The average family might have 2.3 children (or whatever the figure is), but you can't exactly have 0.3 of a child...

Don't Guess...

One final anecdote from [HB13] on the subject of measuring μ. Some smart folks at IBM wanted to know, given the arrival rate λ, what the mean job size, $E[S]$ (which is $1/\mu$) was. Well, $E[S]$ is the mean time required for a job in isolation, so our experiment should be a hundred runs of sending a single job into the system and averaging the

[70] I'm reminded of a funny engineering saying that says if you encounter a system that is nonlinear, you can decide that nonlinear systems are too difficult to reason about, assume the system is linear, and proceed.

values. This is okay, but does not reflect reality where we have things like caching of data and multiple concurrent jobs. There are two basic strategies we can follow for getting a value for μ (depending on whether it is an open or closed system), allowing simple computation of $E[S]$.

The open system strategy is: ramp up λ. Keep piling more jobs on the system. At some point the system will not be able to keep up. Once the completion rate levels off, we hit the limit and we have a value for μ.

The closed system strategy: set it up so there is always work to do. In closed systems, there's often consideration given to *think time*—this is what happens when the user is on the command line and dispatching work to do. The user sends a command and awaits a result. After the result, some time passes while the user decides what to do next (or does code editing before running the compiler again). To keep the system totally busy in the stress test, we need think time to be zero—so additional work is always available. And then we can simply measure the jobs completing per second, giving us μ directly.

practical queueing theory: https://www.youtube.com/watch?v=IPxBKxU8GIQ

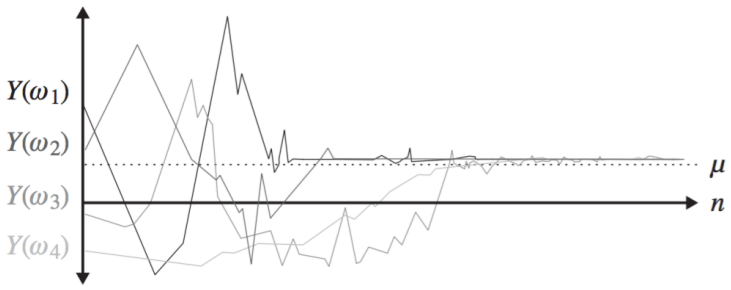
32 — Convergence, Ergodicity, Applications

Convergence

Think back to calculus class. Remember when we talk about limits: $\lim_{x \to \infty} f(x)$. There is an answer for this if the function does (somehow) converge on some value as x grows, e.g. for $f(x) = 1/x^2$ we converge to 0.

We would like to see that our random variables converge. You might flip a coin four times and all four times it comes up heads. That doesn't match our expectation that we should have about half heads and half tails. We have convergence if, given enough samples and enough sample paths, it will converge to the 0.5 we expect. There may be some sample paths that don't converge (e.g., continually coming up heads), but they have a "probability mass" of zero (i.e., they are incredibly unlikely). There are in fact an infinite number of "bad paths", each with probability approaching zero as the number of flips approaches infinity (but that's okay). Probability approaching zero doesn't mean it can't happen, mind you.

An image of what convergence looks like [HB13]:



We won't concern ourselves with systems where there is no convergence. We'll just deal with situations where there is a convergence. Almost every sample path (series of experiments) will eventually behave well if we take enough samples. That is, get past the initial conditions. But sampling is important in our discussion about scalability...

Tim and Enzo

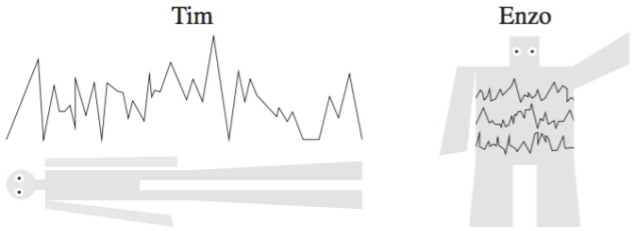
A small but important digression on the subject of sampling, measurement, and testing, from [HB13]. You have an idea of what an average is, but there are two different relevant types of average here—the time average and ensemble average.

Let us just focus on having a single First-Come-First-Serve queue. Every second, a new job arrives with probability p. If there is any work to do, the job being worked on is completed with probability q (and $q > p$). As a definition, let $N(v)$ equal the number of jobs in the system at a time v. In the story, Tim and Enzo are trying to simulate the FCFS system to determine what is the average number of jobs in the system.

Tim decides he's going to run it as one really long simulation. He simulates the queue over a very long period, logging as he goes, taking a million samples. Then he takes the average value over those samples to get the average number of jobs.

Enzo does something slightly different: instead of having one super long simulation, he does 1000 shorter simulations. He waits until the simulation has run for 1000 seconds and then samples the queue at exactly that point, obtaining one value. This experiment is restarted with a new random seed. So after obtaining a thousand samples, he averages these, and Enzo produces another average number of jobs.

A little illustration of Tim and Enzo from [HB13]:



So—who has done this correctly, Tim or Enzo?

The time average has potential problems because we are only looking at a single sequence and maybe something very unusual has happened here in this single run. The ensemble average is more likely what we talk about when we talk about the system being at "steady state" (i.e., past the initial conditions). So we kind of like the Enzo approach. Plus, this is programming for performance (or as a student said, programming for parallelism)—we can do 1000 simulations concurrently if we have enough CPU cores! Tim's approach still has some merit though.

A note about initial conditions: both the Tim and Enzo approaches here require caring about the initial conditions. Enzo needs to make sure that the initial conditions (startup costs etc) have attenuated before the measurement point. Tim needs to ensure that the initial conditions impact a sufficiently small portion of all his measurements.

But! If we have a nicely behaved system, the time average and the ensemble average are the same (so both Tim and Enzo can be correct). What is a nicely behaved system? The word for this is *ergodic*. That probably did not help, so what is an ergodic system? It is a system that is positive recurrent, aperiodic, and irreducible.

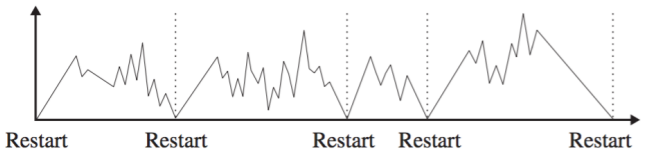
Irreducibility means a process should be able to get from one state to any other state (where state is the number of jobs in the system). This means the initial state of the system does not matter. So if we started at 0 jobs or 10 we could still get to any state in the system (jobs at 2 or 27)...

Positive recurrence means that given an irreducible system, any state i is revisited infinitely often, and the time between visits to that state are finite. So we can define a certain state as being a "restart". The logical choice in the case of a queue or similar is the idea of the queue being empty. Every time the queue gets down to zero jobs, it's a "restart" of sorts.

This is what makes Tim's view and Enzo's view potentially the same. A single long run (Tim's view) is just like a number of independent runs (Enzo's view). Every time we get down to zero jobs in the queue, it's a restart.

The *aperiodicity* condition is required for the ensemble average to make sense or exist. That is to say, the state of the system should not be related to the time; i.e., it is not the case that the system is in state 0 when t is even and state 1 when t is odd. Otherwise the way Enzo chooses to sample, e.g., at $t = 1000$, is potentially going to skew the result (we might get a different answer sampling at $t = 1001$).

A graphical illustration, also from [HB13], that shows how the time average over a single long run can be considered a chain of restarts or "renewals".



Both Tim and Enzo are correct for ergodic systems. Either method works to determine measurements and queueing theory values. Enzo's method has some advantages, e.g. parallelism and the ability to produce confidence intervals.

We've talked about the average number of jobs, but perhaps what we also care about is how long a job spends in the system, on average. We could compute either the time or ensemble average.

$$\text{Time Average} = \lim_{t\to\infty} \frac{\sum_{i=1}^{A(t)} T_i}{A(t)},$$

where $A(t)$ is the number of arrivals by time t, and T_i is arrival i's time in the system. The average is taken over one sample path.

$$\text{Ensemble Average} = \lim_{t\to\infty} E[T_i],$$

where $E[T_i]$ is job i's average time in the system, with the average being taken over all sample paths.

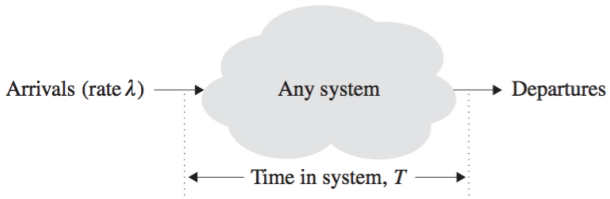
Little's Law

Little's Law is a famous result, saying that the average number of jobs in the system equals the product of the average arrival rate into the system and the average time spent in the system. The source on this section is [HB13].

Open Systems. Let's start with an open system. Here is Little's Law, written more formally:

$$E[N] = \lambda E[T],$$

where $E[N]$ is the expected value of the number of jobs in the system, λ is the average arrival rate into the system, and $E[T]$ is the mean time jobs spend in the system. For example, if a University intakes an average of 5,000 students per year and each student spends an average of 4 years in school, then there are $4 \times 5000 = 20000$ students on average in the University [Sig09]. The basic setup of Little's Law looks something like this [HB13]:



We don't need to know anything about the arrival process (Bernoulli, Poisson, etc. . .), the service time distribution, network topology, etc. It seems intuitive that this is the case (or it should). Imagine a fast food restaurant: they make money by quick turnaround, so they get people out of the place quickly (low $E[T]$) and accordingly they don't require a lot of seating (low $E[N]$). A sit down restaurant is the opposite though; people leave slowly (high $E[T]$) and therefore the restaurant needs lots of seating (more $E[N]$). This example might seem weird from the perspective of the customer though—from your perspective, you may want to enjoy your evening—but the

restaurant is eager to turn your table over, and get you out of there so a new set of guests can be seated. (Another way of looking at this is that restaurants are in the business of renting seats. It's not about food costs.)

If you prefer to think of this in a single FCFS queue version, imagine a customer arrives and sees $E[N]$ jobs ahead of her in the queue. The expected time for each customer to complete is $1/\lambda$, because the average rate of completions is λ. So we can approximate $E[T]$ as being roughly $\frac{1}{\lambda}E[N]$.

Closed Systems. Remember that for closed systems, we have a rule that says there are N jobs in process at any given time (the multiprocessing level of the system). If the system is ergodic, then $N = X \cdot E[T]$ where N is the multiprogramming level, X is the throughput rate, and $E[T]$ is the mean time jobs spend in the system. This assumes that there is zero think time, i.e., that jobs are always ready at once and don't have to wait for silly users.

If we do have to deal with the vagaries of users and think time, then we care more about the response time $E[R]$. So for a terminal-driven system, the expected response time is $E[R] = \frac{N}{X} - E[Z]$ where N is the multiprogramming level, X is the throughput, and $E[Z]$ is the mean time spent thinking.

M/M/1

Probabilistic processes are described according to their models, which will probably be one of the three [Liu09]:

1. Deterministic (D) – The process is predictable and characterized by constant factors. For example, the inter-arrival times are constant (e.g., a task arrives every minute.)
2. Markov (M) – A memoryless process; the future states of the process are independent of the past history. The future state depends on only the present state.
3. General (G) – Completely arbitrary.

We're going to focus on Markov processes, because they are nicer (and we have only limited time). It means that the number of arrivals follow the Poisson distribution; the inter-arrival times follow the exponential distribution; and service times follow the exponential distribution too.

Those letters we saw are part of Kendall notation. It has six symbols, written in a specific order, separated by slashes. The order is $\alpha/\sigma/m/\beta/N/Q$. See the table below for the full explanation:

Symbol	Meaning
α	The type of distribution (Markov, General, Deterministic)
σ	The type of probability distribution for service time
m	Number of servers
β	Buffer size
N	Allowed population size (finite or infinite)
Q	Queueing policy

We often leave off the last three, assuming that there is an infinite buffer, infinite population, and a FIFO queueing policy. If that is the case, then we have only three values. Those three then produce the "M/M/1" and "M/M/k" symbols. "M/M/1" means a Markov arrival process, exponential queueing system, and a single server. When there are k servers, of course the 1 is replaced with the k. These are the systems that we are going to examine.

We should also think about utilization, denoted ρ. It is a fraction between 0 and 1 and it is simply the amount of time that the server is busy. We talked about this earlier in an informal way, but now we can actually calculate it: $\rho = \lambda \times s$ (the arrival rate and service time).

For M/M/1 systems, the completion time average T_q is $\frac{s}{(1-\rho)}$ and the average length of the queue W is $\frac{\rho^2}{1-\rho}$. It is also easy to calculate the waiting time excluding the service time, $T_q' = T_q - s = \frac{s\rho}{(1-\rho)}$.

An example from [Wil13b]: we have a server that completes a request, on average, in 10 ms. The time to complete a request is exponentially distributed. Over a period of 30 minutes, 117 000 jobs arrive. So this is a M/M/1 situation. How long did it take to complete the average request? What is the average length of the queue?

The service time s is given as $0.01s$ and the arrival rate is 65 requests per second. So we can calculate $\rho = 0.01 \times 65 = 0.65$. So we have what we need to plug and chug using the formulæ from above to find the time to complete the average request is 28.6 ms and the average length of the queue is 1.21.

What about the number of jobs in the system? The value $\frac{\rho}{1-\rho}$ gives the average number of jobs, including the waiting jobs and the ones being served. It is an average, of course. The probability that there are exactly x jobs in the system at any time is given by the formula: $(1-\rho)\rho^x$. The probability that the number of jobs is less than or equal to n is then given by: $\sum_{i=0}^{n}(1-\rho)\rho^i$ (the sum of the probabilities of each of the numbers from 0 up to n). If you want to know the probability that there are more then n at a time, then you can compute the sum from $n+1$ up to infinity. That might be unpleasant to calculate, but remember that probabilities sum to 1, so you can say that the probability of more than n requests at once is simply $1 - \sum_{i=0}^{n}(1-\rho)\rho^i$.

M/M/k

Now let us take it to multiple servers (k of them, in fact). We will say jobs arrive at a single queue and then when a server is ready it will take the first job from the front of the queue. The servers are identical and jobs can be served by any server. So far, so simple.

Sadly, the math just got harder. Let's turn again to [Wil13b] as the source for this section. The server utilization for the server farm is now $\rho = \lambda s/N$ (the average utilization for all N servers). To make our calculations a little easier, we want an intermediate value K which looks scary, but is not so bad:

$$K = \frac{\sum_{i=0}^{N-1} \frac{(\lambda s)^i}{i!}}{\sum_{i=0}^{N} \frac{(\lambda s)^i}{i!}}.$$

The first term, $i = 0$, is always 1. The denominator is always larger than the numerator, so K is always less than 1. K has no intrinsic meaning, it is just a a computational shorthand so the other formulæ are not so messy.

What is the probability that all servers are busy? We represent this as C, the probability a new job will have to wait in the queue.

$$C = \frac{1-K}{1-\frac{\lambda s K}{N}}.$$

The M/M/k formulæ, then, for the average completion time and average length of the queue are:

$$T_q = \frac{Cs}{k(1-\rho)} + s \qquad \text{and} \qquad W = C\frac{\rho}{1-\rho}.$$

Let's do an example. Suppose we have a printer that can complete an average print job in two minutes. Every 2.5 minutes, a user submits a job to the printer. How long does it take to get the print job on average? We're starting with a single printer, so the system is M/M/1. Service time s is 2 minutes; the arrival rate λ is $1/2.5 = 0.4$. So $\rho = \lambda \times s = 0.4 \times 2 = 0.8$. So $T_q = s/(1-\rho) = 2/(1-0.8) = 10$. Ten minutes to get the print job. Ouch.

Here we have an opportunity to use the predictive power of queueing theory. Management is convinced that ten minute waits for print jobs is unreasonable, so we have been asked to decide what to do: should we buy a second

printer of the same speed, or should we sell the old one and buy a printer that is double the speed?

The faster printer calculation is easy enough. Now $s = 1.0$ and λ remains 0.4, making $\rho = 0.4$. So rerunning the calculation: $T_q = s/(1-\rho) = 1/(1-0.4) = 1.67$. 1:40 is a lot less time than 10:00!

The two printer solution is more complicated. So let us calculate K as the intermediate value.

$$K = \frac{\sum_{i=0}^{N-1} \frac{(\lambda s)^i}{i!}}{\sum_{i=0}^{N} \frac{(\lambda s)^i}{i!}} = \frac{\frac{(\lambda s)^0}{0!} + \frac{(\lambda s)^1}{1!}}{\frac{(\lambda s)^0}{0!} + \frac{(\lambda s)^1}{1!} + \frac{(\lambda s)^2}{2!}} = 0.849057.$$

Now we can calculate C as 0.22857 and T_q as 2.38 minutes (by simple plug and chug calculations given the formulæ above). Two observations jump out at us: (1) we doubled the number of printers, but now jobs are completed almost four times faster; and (2) the single fast printer is better, if utilization is low.

That is an important condition: if utilization is low. At some point will the two printers be a better choice than the single fast one? What if both printers are used to the max (100% load)...?

Queuing for Performance

The plan is to take queueing theory and apply it in a performance model. The guide to this section is [Wil13c]. The basic process is:

1. Convert to common time units.
2. Calculate the visitation ratios V_i.
3. Calculate the device utilization ρ_i.
4. Calculate the CPU service time.
5. Calculate the device time.
6. Find the bottleneck device.
7. Calculate the maximum transaction rate.
8. Calculate the average transaction time.

Let us execute this process on a web server system that serves 9 000 pages per hour. Here are the known values:

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Webpages	9 000					
CPU					42%	
Disk 1	108 000		11ms			
Disk 2	72 000		16ms			
Network	18 000		23ms			

Step one is to convert to common time units; in this case, seconds. That would be a simple and common time unit. Let's also look at the λ values - reported counts divided by seconds in the reporting period.

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Webpages	9 000	2.5				
CPU					42%	
Disk 1	108 000	30	0.011s			
Disk 2	72 000	20	0.016s			
Network	18 000	5	0.023s			

The visitation ratio is the number of times a device is used in each transaction; divide use by number of transactions to get V_i (you could also log this sort of thing). The visitation ratio of the CPU is the sum of all other visitation ratios. Why? Suppose we do a disk read: the disk is visited, and when the disk read completes, we go back to the CPU and it picks up the data that's just been shuttled in from the impossibly slow disk.

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Webpages	9 000	2.5		1		
CPU	207 000	57.5		23	42%	
Disk 1	108 000	30	0.011s	12		
Disk 2	72 000	20	0.016s	8		
Network	18 000	5	0.023s	2		

Next, calculate device utilization: $\rho = \lambda \times s$. That is, arrival rate times service time.

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Webpages	9 000	2.5		1		
CPU	207 000	57.5		23	0.42	
Disk 1	108 000	30	0.011s	12	0.33	
Disk 2	72 000	20	0.016s	8	0.32	
Network	18 000	5	0.023s	2	0.115	

A small oddity: in the CPU we have a percentage for utilization rather than a decimal number. Just convert it to 0.42. And we can also get the service time of the CPU by rearrangement of the utilization formula to $s = \rho/\lambda$.

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Webpages	9 000	2.5		1		
CPU	207 000	57.5	0.0073s	23	0.42	
Disk 1	108 000	30	0.011s	12	0.33	
Disk 2	72 000	20	0.016s	8	0.32	
Network	18 000	5	0.023s	2	0.115	

And the device time is the final thing we can fill in for this table: $V_i \times S_i$ (just like the column header says!).

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Webpages	9 000	2.5		1		
CPU	207 000	57.5	0.0073s	23	0.42	0.168
Disk 1	108 000	30	0.011s	12	0.33	0.132
Disk 2	72 000	20	0.016s	8	0.32	0.128
Network	18 000	5	0.023s	2	0.115	0.046

Did we need to complete the whole table? Probably not. In a practical sense what we cared about the most was the ρ column—utilization. The bottleneck device, i.e., the one that limits our maximum throughput, is the one that is the busiest. Thus, the one with the largest utilization. This application appears to be CPU bound; it has the highest utilization at 42%, well ahead of disk 1 and disk 2.

Having identified the bottleneck device as the CPU, we can make a prediction about the maximum rate of transactions (web page requests) we can serve: $\frac{1}{S_i V_i}$ or in this example, 5.95. This is also called saturation. If λ exceeds this saturation point, we will not be able to keep up with incoming requests.

With this table we can also calculate the average transaction time: it is the sum of the $S_i V_i$ columns. In this example, it is 0.474 seconds.

It Gets Worse The typical assumption is that we know the service times for each device. Unfortunately this is not true; usually performance monitoring gives us the average size of a device queue. So we had better apply queuing theory here. Once again, credit to [Wil13a] for the section.

The average size of a device's queue is W, and for a queue with characteristics M/M/1 then $W = \frac{\rho^2}{1-\rho}$. Combining the known W with the average arrival rate λ, we can work out the service time. $W = \frac{(\lambda s)^2}{1-\lambda s}$, so:

$$s = \frac{-w \pm \sqrt{w^2 + 4w}}{2\lambda}.$$

Yup. The quadratic formula strikes back.

33 — More Advanced Queueing Theory

New Considerations

There are a few new considerations, or, if you prefer, complications to queueing theory that we haven't yet covered in the fairly simple discussion of queueing theory we have had so far. But real life is considerably more complicated than a simple model. We'll talk about two settings, one that I love and one that I hate: food halls and Service Ontario. No points for guessing which is which.

If you're not familiar with them, a quick refresher. A food hall, or food court, is a place where there are a number of counter-service restaurants that frequently offer different cuisines. Service Ontario is a place of interaction with the administrative state, specifically the services administered by the provincial government of Ontario.

Multiple Services. The way we have discussed the idea of services is that all the offered services are the same, or at the very least, every server can deliver every kind of service. That's sometimes reasonable—at Service Ontario, there are many different services available (driver's license renewal, health card renewal, vehicle registration update, etc) and any one of the staff members there can help you with any one of those services[71].

There are, of course, other situations where an individual member of staff cannot provide all services and therefore you must queue for the specific thing that you want, even if the other queues are shorter or empty. A food hall works like this. The Mexican restaurant may be very popular because they have excellent tacos, resulting in a long queue for that particular place. There may be no queue at the Gelato stand, but that doesn't help; the Gelato place cannot provide tacos. And while both of these are food, you will probably agree that they're not (always) interchangeable. If you really want tacos, gelato won't do.

Maintenance, Planned and Otherwise. In real life, services will potentially have downtime, planned or otherwise, which can throw a wrench in things unexpectedly and sometimes that means closing the queue and sending requesters elsewhere. If the pizza place's oven breaks, they can't make any more pizza, and everyone in line for the pizza needs to go somewhere else if they want to eat today. We don't usually account for this directly when planning and designing our systems, possibly because we expect things to always be available and for people to be paged to fix it if it's not (see our next topic for more about this). But we might account for regular maintenance or planned downtime in our estimates of how many customers we can serve in a given period.

In principle, new services can also come online also. I don't mean more workers at Service Ontario, but more like what happens if a new restaurant opens. If a new Banh Mi place opens then people who are currently in line for a different food might prefer to switch because they like that cuisine better. In reality, like a restaurant opening, the presence of a new kind of service is rarely a true surprise: people have worked hard to build, test, and deploy it and it's not as though we just wake up one day and suddenly a new thing is just... there.

Interchangeability. We already hit on the idea of interchangeability when talking about the food hall example. It's a spectrum: sometimes things are totally interchangeable, sometimes partly, and sometimes not at all. In the food hall example, total interchangeability is what happens if you're just extremely hungry and you would be completely happy whether you had tacos, pizza, or shawarma. Partial interchangeability can happen when you have preferences but would accept something else: you want tacos but would choose shawarma under certain

[71] Though not necessarily in the official language of Canada of your choice—`https://www.ledroit.com/actualites/politique/2024/06/14/services-en-francais-on-nous-prend-pour-une-erreur-administrative-K5OSXVISMRFCDMXOFURHZOMUQQ/`

circumstances. And no interchangeability happens when you have your heart truly set on pizza and will accept nothing else.

Whether things are interchangeable depends on the nature of the services and the needs of the people requesting them. Just as there are different services in this more complicated world, requesters also behave differently.

All of the above food hall examples are about food, so there's at least a possibility of interchangeability because every option will have some nutritional value (in the sense of being edible food, produced by a place which has passed health inspections, with calories in it; a nutritionist might say the gelato has no nutritional value because it's not a "healthy" choice). If you went to Service Ontario to renew your driver's license, there are really no alternatives that get you the same result: a new health card just isn't the same thing.

Then there are the needs of the person requesting the thing: If someone is a vegetarian, they might be okay with tacos or pizza, but not shawarma. If they are a vegan, maybe the only option for them is the taco stand, no matter how good the pizza place is or how long the line for tacos is.

Too Long. While we're on the subject, when you're in the food hall and the length of a particular queue is too long, we may experience one of three behaviours: balking, reneging, or "loss".

Balking is what happens when you look at the line and you decide it's too long and there's no point in even getting into the line. That's very common: if I want to go to Service Ontario and it's so busy[72] that there is a long line out the door, I'm not going to bother. I'll come back another time (if I can—sometimes it's important to do something before a deadline). In the food hall, if the queue for tacos is really long, I'll line up for the shawarma.

Reneging is what happens if I enter the queue for tacos, but before I get to the front of the queue, I give up (leave the queue). If it's taking too long and I'm really hungry (or just impatient), this could happen. It could also happen if I'm at Service Ontario in between things (e.g., on a lunch break) and if time runs out I need to go back to work.

Loss is what happens when service is refused because of capacity limits. This comes from the idea of the early telephone systems, where there were k circuits and if none of the k circuits was free at the time a person wanted to make a call, that call was just dropped (that is, the call fails) [HB13]. That makes it a M/M/k/k system, in queueing theory notation. In the food hall or Service Ontario example, this is what happens if there's a capacity limit to the building and I'll be turned away by security if the building is full. I'm not allowed to queue up, no matter how badly I'd like to.

In both cases where I choose to leave, there's an implicit or explicit estimation of the waiting time provided. When I choose not to enter the queue at all, it's because I've looked at the queue length, and maybe done some assessment of the service time, and calculated that the wait time exceeds my willingness to wait. It's also possible that the establishment is kind enough to put out signs that say that the expected waiting time from this point is X minutes. Not very common in a food hall or Service Ontario, but maybe the case in other places.

Obviously, my initial estimate can be wrong if I guessed incorrectly about the service time, or if the line is so long I can't see all of it and I don't get a good estimate. But another reason why my estimate might be wrong is if people can join the line ahead of me. Wait, what?

Priority. When I was last at Service Ontario, an elderly person with mobility restrictions showed up while I was waiting in line and that person was permitted to go to the front of the queue. It's sensible that priority would be given to this person – asking them to stand in line a very long time is not nice. But of course, when they go into the line ahead of me, it increases my wait time. This increase may result in my decision to leave the line as the time has increased to the point that I can no longer, or at least no longer wish to, wait.

Priorities for some groups over others are actually really interesting, because they have interesting effects and open up questions:

- How much, if any, does giving priority to one group over another help the group being given priority?
- How much, if any, does giving priority to one group disadvantage the group not being given priority?

[72] pro tip: go at opening time, I (PL) have been in and out in 8 minutes.

- Can the priority system incentivize people to choose things that are less popular?
- Recognizing that if everyone has priority, nobody has priority, how many requests can have priority before all benefit is lost?

Laboratory Study with a Mouse

This section relies on a rather informative video by the channel Defunctland, about the history of the Disney Fastpass system [Per21]. The actual video contains a lot of discussion about the history of Walt Disney World and other things that are not relevant here. But there are a few interesting things we can learn from it. In the video, there's a lot of background info but also some explanation of the simulation that was used to evaluate the situation.

Simulation? Yes—at some point the queuing theory problem becomes so complex that our ability to reason about it and do the math with spreadsheets or Wolfram Alpha or hand calculations is a limiting factor. And a simulation allows us to validate our theories. But why is this system so complex that it requires a simulation?

If we abstract away some of the details of the Mouse and his friends, we're left with a system has many details that we need to concern ourselves with:

- Every customer (requester) is an independent agent, which implies:
  - They have different times of arrival at and departure from the park. With that said, most arrive early in the day and relatively few arrive later in the day
  - They have different preferences of what they want to do while there
  - They have different willingness to wait for the things they want to do
  - They may or may not be willing to come back another day
- The park has opening and closing times which implies:
  - Requests cannot be submitted before opening time
  - Requests cannot be submitted after closing time
  - Requests submitted too close to closing time may not be served before closing
- The park has different services that each have their own service rate and any of them could be down for maintenance (independently of any others)
- The services have a fixed maximum capacity: you cannot make more seats on the rides or run them faster to get more people through quicker

Relevance? An important observation here is that this model is fairly different from what we are used to talking about when thinking of servicing requests. When I go to Service Ontario, I want to renew my license as fast as possible and leave as fast as possible and hope not to return any time soon. If I am at the food hall, I may want to sample some number of different cuisines, but I'll eventually be full (or at least be unable to eat any more food) and then I'll leave. But in this model, the tourist goes to the Mouse Park and stays for some period of time, trying to do as much as they want to do. That doesn't sound like most of our software service scenarios. Is anything we're learning in this model applicable?

I'll argue yes. If I'm a person waiting in line for a specific ride, it's entirely irrelevant to me whether the 400 people in front of me have just arrived at the park, or if they've been here for hours and this is their tenth ride. Rides don't have to be completed in a specific order (in the sense that nothing bad happens if someone does them out of order).

You could just imagine that this model is functionally equivalent to one in which each person who goes on a ride immediately leaves the park, never to return, and is instantly replaced by another guest who has the same preferences about what to do. With more and more complex priority systems, this way of thinking about it might not hold up, but hopefully this argument is convincing.

User types. The simulation has a few different archetypes for the users that represent their different "personalities" based on the expected type of person who would attend the park. Some of them come frequently because they have an annual pass, so they can always come back another time if they prefer. Others are here today and today only and don't want to miss out. But without getting bogged down in the archetypes, the type of user decides (1) how long they will stay at the park, (2) when they will balk (what is their willingness to wait for a particular ride), and (3) what they want to do while they're here. The third point covers both their preferences of what rides they want to go on (in what priority sequence) and if they want to do other things in the park (things that aren't rides).

Priority Systems. To establish a baseline, the simulation has an option where there's no priority system: everyone is equal. Then there are two different kinds of priority systems which the Mouse calls 'FastPass" and "FastPass+" (I'm sure the developers wanted to call it FastPass^{++}).

In the no-priority system, everyone is equal. Wait times are just based on how popular things are and nobody gets to cut in line. It helps wait times to be predictable, because the line a person in will advance at a fairly steady rate. No priority may seem fair at a glance, but is it optimal? Let's see.

In the FastPass system, instead of waiting in line, a person could get a little ticket that specifies a return time, and when they return they can get to the front of the line. This allows the people waiting to do something useful or fun in the meantime (and maybe profitable if you buy some food). That just makes it a virtual queue system, as far as we're concerned, and not really a priority lane. You're waiting in the queue just as long as you would otherwise, but you're not having to stand in the physical one doing nothing while you wait. This does allow you to potentially get some other stuff done (imagine you can get some gelato while waiting your turn for pizza). The FastPass system is applied only to the (few) most popular things, so for all other attractions there is no fast lane.

In the FastPass^{++} system, in advance of going to the park (or on arrival), you get three priority passes to some specific attractions well in advance. The passes can be "sold out" if there is sufficient demand. This resembles making reservations more so than waiting in a virtual queue. As you can imagine, people try really hard to get the priority passes for the most popular things.

Regardless of whether FastPass or FastPass^{++} there are two queues for any individual service: the priority queue (or fast lane) and the standby queue (that's the one for everyone else). If a ride can seat n people at a time, the ratio of priority to standby is important. Under typical circumstances the ratio of priority to standby is something like 4 priority to every 1 standby guest. In times of big backlog in the express line, it can be 20:1 or even 100:1.

Results

So let's recap the results from the simulation [Per21]. We're looking at the results in the video, but... did you want to play around with the simulation yourself? `https://github.com/TouringPlans/shapeland` – it's python, but it's not super complicated code. The results are broken down into (1) standby waits, (2) overall waits, and (3) average number of rides experienced.

Standby Waits. Standby waits increase using FastPass, but that's not super surprising: if some people can cut to the front of the line, it delays everyone else. The FastPass^{++} approach increases the wait times on every ride except the most popular one (even though it's a small amount), often by a significant amount. Okay, so we know that it makes stuff worse in standby, but not everyone is in the standby line. So what's the overall impact?

Overall Waits. With no priority system, if a guest tried to do everything once, the average wait time would be about 41 minutes, but because people prefer to do the most popular things, the wait is about 58 minutes (on average) in reality. That's the baseline.

With FastPass, average wait in standby if doing every ride is 48 minutes, but actual wait time is more like 40 minutes—about 2/3 that of the no-priority system. That sounds pretty good! But note that the benefit here is coming from pushing people to less popular attractions which they can do while waiting for a more popular thing.

The FastPass^{++} solution raises the average standby wait for doing everything to 67 minutes but the typical wait per attraction is 42 minutes. Again, this is because the wait times are longer on standby. But, average times are reduced, because people are encouraged to go to less popular things to make use of those.

Average Rides Ah, you've probably figured out at this point that one of the real goals of these priority systems are to incentivize people to use the under-utilized attractions. The average number of rides without any priority system is 3.31; with FastPass it's 3.77; and with FastPass^{++} it's 4.23.

The distribution with FastPass^{++} is quite uneven though: it increases the number of people who go on many rides, and also the number of people who go on very few. So now we have winners and losers in the system, whereas FastPass does not seem to have this same effect of increasing inequality: some people do many rides in the day, but a lot more people get zero or one.

Lessons and Limitations

So the bottom line is: giving priority passes doesn't have a big impact on the wait times for the most popular attractions, but it does encourage better utilization of the less-popular things. And more usage overall.

If we didn't speed up wait times for people on rides generally, only encouraged people to do other things too, is that improvement? The Mouse is in the business of entertainment. The megacorporation wants you to spend money with them (consume, obey) and most likely a guest will return and spend more money if they have enjoyed their visit. Is going on more rides more enjoyable than more time in queues?

Maybe! That's at least what I think. It probably make some intuitive sense that at one extreme end of the spectrum where I never have to wait at all, I'm extremely happy; at the other extreme end where I wait in line all day and never ride anything, I'm very unhappy. I'm also pretty unhappy if I don't get to do my favourite things even if other things are more available. Still, it seems like getting to do more rides is good for increasing guest happiness if most guests follow a similar evaluation process.

To my knowledge, the simulations didn't account for downtime during the day. There may be some maintenance baked into the estimates of the capacity, but I mean the kind of downtime where a particular ride just goes offline and won't be back that day. In a simple model of that scenario we could just send everyone in the queue back to making a decision about what to do (go to next attraction, go home, etc). But in reality people who has a FastPass for the ride that went down might get another FastPass as compensation, which may have weird downstream effects. Also, people who were in the standby queue for a long time may get a compensatory FastPass for another attraction too, which would cause some chaos by increasing the number of passes in circulation. That might be a fun extension of the simulation.

Inequality is increased even more in real life than in the simulation by the knowledge factor. People have learned the nuances of the system and those who know the secrets (or at least learn about enough of them) get to do more things. If you have mastered the system, you get to do a lot more rides (8–9!) than someone completely unaware of it who might only get 1–2 rides in. There are countless videos out there about how to take advantage of it.

A final lesson based on system nuances is that the more complex your system is, the more ways there may be for people with expert knowledge to exploit it and benefit themselves at the expense of others. Such unexpected user behaviour can really mess up your capacity planning and make the user experience for everyone else worse.

34 — DevOps: Configuration

DevOps for P4P

So far, we've talked almost exclusively about one-off computations: you want to figure out the answer to a question, and you write code to do that and when it's finished, you are done. The course assignments have been like that, for instance. There are lots of programs in the world that are solely executed on-demand. Still, a lot of the time we want to keep systems running over time. That is, a service that is available (if not always then close to it) and responds to requests whenever they may happen to arrive. That gets us into the notion of operations. Your service or product is more likely than ever to have at least some component that's server side (whether hosted in a cloud service or not) that's under your control. And that server side component is something you want to keep running as it's supposed to be "generally available".

The theme in this topic will be using software development skills in operations (e.g., system administration, database management, etc). This does have some relevance, because the operations (or IT, if you prefer) processes and procedures, while different from development, have some similarities.

Even when we've talked about multi-computer tools like cloud computing, it still has not been in the context of keeping your systems operational over longer timescales. The trend today is away from strict separation between a development team, which writes the software, and an operations team, which runs the software and infrastructure.

The separation is totally nonexistent at the typical startup company. There isn't the money to pay for separate developers and operations teams. And in the beginning there's probably not that many servers, just a few demo systems, test systems, etc... but it spirals out from there. You're not really going to ask the sales team to manage these servers, are you? So, there's DevOps, also sometimes called Software Reliability Engineering.

Is DevOps a good idea? Like most ideas it can be used for both good and evil. There's a lot to be said for letting the developers be involved in all the parts of the software from development to deployment to management to training the customers. Developers can learn a lot by having to do these kinds of things, and be motivated to make proper management and maintenance tools and procedures. If we make the pain of operations felt by developers, they might do something about it. If it's the problem of another team, somehow those tickets just never make it to the top of the backlog.

As the company grows, you might think about whether a dedicated operations team is preferable. That may be sensible, but shifting all workload to the operations team probably isn't ideal for the reasons above. And it is probably not scalable, either. If the operations team has to be involved in everything they can quickly become a bottleneck and everyone is happier if the development teams are able to solve their own problems instead of opening tickets and then hassling people to please just do the thing.

Continuous Integration. This is now a best practice – each change (or related group of changes) is built and tested to evaluate it. Once upon a time, putting the code changes together didn't happen on every build, but nightly or otherwise. That was a weird time when builds were slow and expensive. I think we're past that now, especially given that we have use of version control, good tests, and scripted deployments. It works like this:

- pull code from version control;
- build;
- run tests;
- report results.

What's also key is a social convention to not break the build. These things get done automatically on every commit and the results are sent to people by e-mail, Slack, Teams, or whatever you use.

Thanks to Chris Jones and Niall Murphy for some inputs for following points.

Configuration as Code

Systems have long come with complicated configuration options. Sendmail is particularly notorious (though who runs their own mail server anymore?), but apache and nginx aren't super easy to configure either. But the environment that you're running your code in is also a kind of configuration. Furthermore, it's an excellent idea to have tools for configuration. It's not enough to just have a wiki page or github document titled "How to Install AwesomeApp" (fill in name of program here). Complicated means mistakes and people will forget steps. Don't let them make mistakes: make it automatic. The first principle is to treat *configuration as code*. Therefore:

- use version control on your configuration.
- implement code reviews on changes to the configuration.
- test your configurations: that means that you check that they generate expected files, or that they spawn expected services. (Behaviours, or outcomes.) Also, configurations should "converge". Unlike code, they might not terminate; we're talking indefinitely-running services, after all. But the CPU usage should go down after a while, for instance.
- aim for a suite of modular services that integrate together smoothly.
- refactor configuration files (Puppet manifests, Chef recipes, etc);
- use continuous builds.

One particular example of applying all those principles to infrastructure is Terraform. Its whole purpose is to manage your config as codes situation where you want to run your code using a cloud provider (e.g., AWS), then you can control the infrastructure using Terraform: you write the configuration files that say you want this service with these permissions (and any other details) and then you can apply that configuration easily. Even beyond that, you can ask Terraform to manage things like who has access to your GitHub repositories and who is in what groups (e.g., reviewers).

Terraform does support a *plan* operation so it can tell you what it will do, so you can verify that, before anything is actually changed. The plan can also tell you expected changes in terms of cost, which both helps verify that we aren't about to give all our money to Jeff Bezos but also that a small change is actually small. If you are happy with the change, *apply* it!

The plan operation isn't perfect as things can change between the plan and apply steps, and some things like unique identifiers are really only known if they are created. Non-destructive changes are generally easy to deal with; just make another PR that corrects it. Destructive changes, however...

It's easy for very bad things to happen with Terraform as well: you could accidentally tell it you want to destroy all GitHub groups and it will gladly carry it out. This has the side effect of causing some people to message you on Slack in a panic, thinking that the removal of their GitHub access is actually a sign they are being fired. They were not. But I see why they were worried, honestly. Restoring some information in destructive changes might not be as easy as just reverting the change: if you told your tool to destroy a database, reverting the change will re-create the database, but not its contents. You took backups, right?

Common Infrastructure

Using tools to manage the infrastructure is a good start, but it also matters how services use it. You should view different parts of your infrastructure as having an interface. Communication is done exclusively via the interface or API. This reduces the coupling between different components, and allows you to scale the parts that need scaling.

Try to avoid not-invented-here syndrome: it is usually better to use an existing tool—whether open-source, commercial, or provided by your cloud platform—than to roll your own. Some examples might be:

- Storage: some sort of access layer (e.g., MongoDB or S3);
- Naming and discovery (e.g., Consul)
- Monitoring (e.g., Prometheus)

However, be prepared to build your own tools if needed. Sometimes what you want, or need, doesn't exist (yet). Think carefully about whether this service that is needed is really part of your core competence and whether creating it adds sufficient value to the business. It's fun to make your own system and all, but are you doing what you're best at?

Think extra carefully if you plan to do roll-your-own anything that is security or encryption related. I'm just going to say that unless you have experts on staff who know the subject really well and you're willing to pay for external audits and the like, you're more likely to end up with a terrible security breach than a terrific secure system.

As a second followup soapbox point to that: if what you are looking for doesn't exist, there might be a reason, Maybe the reason is that you are the first to think of it, but consider the possibility that it's not that good of an idea (either due to inefficiency or just not being great in principle).

With that said, big platforms like AWS are constantly launching new tools that might do what you want and the best strategy may be just wait until the managed service is provided for you. Patience can be rewarded, in this regard, and you can easily feel very frustrated by investing a lot of effort into planning, building, and launching a feature only to find that your provider has rendered it redundant immediately thereafter.

Naming

Naming is one of the hard problems in computing. There is a saying that there are only two hard things in computers: cache invalidation, naming things, and off by one errors. There are a lot of ways to name things. Naming is necessary for resources of all kinds. There's the Java package approach of infinite dots for your server: live.application.customer.webdomain.com or however you want to call it. Whatever we pick, though, names need to be consistent or at least close to it.

Debates often rage in companies about whether teams, services, or anything else can or should have names that are "meaningful" or not. There's arguments for both sides: if the service is called `billing` it may be helpful in determining what it does, more so than if it were called `potato`. But there's the possibility of confusion around whether when you say the word billing you mean the service or the team. And what if we want to replace the billing service with a new one? Is it `billing2`? All of this presupposes, also, that your service does exactly one thing. What if the service is "billing and accounting"; do you rename it? Is accounting just folded into billing? How do we even decide such things?

Allegedly-descriptive names aren't always the easiest to figure out either. I've seen examples where the teams are called (anonymized a bit) "X infrastructure" and "X operations" and I'd estimate that 35% of queries to each team result in a reply that says that the question should go to the other team. It gets worse when a team is responsible for a common or shared component (e.g., library).

The *real* solution to this kind of problem at least in my opinion, is similar to the idea of service discovery: we need a tool that provides directory information: if I want to know about `potato` I need to be able to look it up and have it send me to the right place. Tools for this, like OpsLevel, exist (even if they do much more than this). Such tools can also give some information about service maturity—are you using deprecated things, do you have unpatched security vulnerabilities, is there enough test coverage...?

There are potential morale implications for insisting on boring names for teams and services. A team that has named itself after some mythological name or fictional organization can have some feeling of identity in it—Avengers, Assemble—and that can be valuable.

Servers as cattle, not pets

By servers, I mean servers, or virtual machines, or containers. It's much better to have a reproducible process for deployment of a server than doing it manually every single time. The amount of manual intervention should be minimized and ideally zero. If this is done you can save a lot of hours of time, reduce errors, and allow for automatic scaling (starting and stopping servers depending on demand).

The title references the idea that cattle are dealt with as a herd: you try to get the whole group to move along and do what they need. Pets are individuals, though, and you'll treat them all differently. This amount of individual attention quickly becomes unmanageable and there's no reason why you should worry about these differences in a world with virtualization (containers) or similar.

As with managing infrastructure, I'll give a specific example: Kubernetes. This is used to automate deploying and scaling of applications. This means you don't have to manually manage them; if you want to deploy a new version of a container, just tell Kubernetes to do it. The part about how it does so is that it provides a framework for your environment. It can do load balancing, automatic rollouts and reverts (if something went wrong), will figure out what instances run on what hardware, check the health of instances, replace them if they die, and more.

Canarying. Rather than just deploying all at once, one thing we could do to test a deployment is referred to as canarying. There are two ways this can happen. One is to deploy new software alongside the existing software and redirect a small fraction of traffic to the new service and evaluate it. If it's good, keep or increase the fraction of traffic; otherwise, take it out of service. The other way of canarying is upgrading some, but not all, instances of the service and checking if they are working as expected.

This is also called "test in prod". Sometimes you just don't know how code is really going to work until you try it. After, of course, you use your best efforts to make sure the code is good. But real life is rarely like the test system. I've seen many operations that work beautifully in the development environment where there are 100 000 records... and time out in production where there are 10 000 000. But for canarying deployments of the second kind, the basic steps:

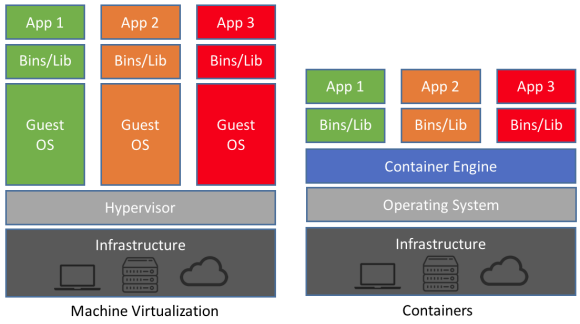
- stage for deployment;
- remove canary servers from service;
- upgrade canary servers;
- run automatic tests on upgraded canaries;
- reintroduce canary servers into service;
- see how it goes!

Of course, you should implement your system so that rollback is possible. If the canary deployment results in changes to the database table structure and this database is shared amongst all servers, trying to deploy the change can break the old servers and leave a rollback implausible because we'd have to write a reverse database migration to undo it... if that's even possible. Because sometimes a migration is destructive!

Containers. What about containers? Well, let's see how we got there first. In the beginning, you had services where you installed the binaries and config files by hand. That sucked. So there were packages; a package includes everything the program needs (including a list of dependencies) and a script to install it and set it up. Great! But if you just install multiple services on the same machine you don't get isolation and you might have incompatible versions of dependencies and you're in RPM hell (see also: JAR hell, classloader hell, and DLL hell).

Right, so instead you say you should have virtual machines: you configure the VM parameters and install the guest OS and set it up (and you can copy-paste the initial image, which helps) but for every application you have a guest operating system running underneath. Maybe we don't need every app to have its own guest OS; why do we have to install the same security patch ten times...?

Containerization gives many of the advantages of this separation, but without nearly so much overhead of the guest operating systems (both its maintenance and runtime costs). Containers are run by a container engine so there is some abstraction of the underlying hardware, and the container is assembled from a specification that says what libraries, tools, etc. are needed. And thus when the container is built and deployed it is sufficiently isolated but shares (in read only mode) where it can. So a container is a very lightweight VM, in some sense. See this diagram from [Cha18]:



So the goal of your build process should be to produce a container that is ready to be deployed and managed by the other parts of your infrastructure (e.g., Kubernetes).

Containers do potentially come with a downside of letting you just not upgrade things. Whereas before if you installed the OS patches you might get updates in your shared libraries, the container allows you to keep an old insecure version of a library around forever. So, don't do that. Keeping on top of security updates is important. Let's talk about that, actually, in the next course topic.

35 — DevOps: Operations

DevOps, but Operations

Let's imagine that we've got our services up and running and doing The Thing that they are supposed to do. A good start, but we need to keep it running. How do we know if there's a problem? Monitoring, of course. And if we see a problem? Alerting.

Monitoring and Alerting

Monitoring is surprisingly difficult. There are a lot of recommendations about what to monitor and what to do about it. Applications should have health checks, though these are often basic: is the app running and able to respond to incoming requests? A more realistic health check should probably see that the system can actually do useful work, such as reach its database. You can even expand on this by automated testing of basic workflows, like users can log in and see their data.

A simple health check that verifies that the app can respond or reach its database won't show us performance problems. Yes, the app might be able to reach the database and the automated test user may be able to log in, but what if these are ten times slower than usual? Here are some things we could look at:

- CPU Load
- Memory Utilization
- Disk Space
- Disk I/O
- Network Traffic
- Clock Skew
- Queue lengths
- Application Response Times

We've already covered a number of ideas about profiling (and queueing theory!) that help us understand what and how to measure in many of these areas, so we won't repeat it here.

With multiple systems, you will want some sort of dashboard that gives an overview of all the systems in a summary. The summary needs to be sufficiently detailed that you can detect if anything is wrong, but not an overwhelming wall of data. Information dashboard design is a complicated topic and the subject of some systems design engineering courses. It's a little bit late for me to recommend you take one of those as we're at the end of the course, but maybe for graduate studies or continuing education?

Realistically, you do not want to pay someone to stare at the dashboard and press the "Red Alert!" button if anything goes out of some preset range of what is okay. No, for that we need some automatic monitoring. Monitoring tools allow us to set up alerting based on certain conditions or thresholds. Some immediate examples:

- CPU usage exceeding threshold for a certain period of time
- Increased rate of error logs over a period of time
- A service has restarted many times recently
- Queue length very long

- Taking too long to complete a workflow

There is a spectrum here of what these mean... CPU usage being high may not be a problem on its own. If the system is at 80% CPU usage consistently, it may be keeping up with the workload and it's just an indication that we're close to the maximum here. If it's at 100% then we should be worried that our performance is limited by CPU availability. If queue lengths are long, it might only be a problem if a certain response time is expected – merely having a big backlog of work to do isn't a problem unless it means missing deadlines or being unable to do other things. Other measures are more outcomes-focused: if logging in takes too long, users are going to struggle or be frustrated, so we should consider it a problem.

Remember also the lesson from Sherlock Holmes in "The Adventure of Silver Blaze"—the dog that did *not* bark was a clue as to who did the crime. Setting minimum thresholds for workflows may identify problems as well; if nobody has logged in to your application for an unusually long time, that might be a sign that something is wrong with the login process. On the other hand, it might be New Year's Day on a Sunday, so maybe nobody's logging in because they're not working today.

The final option for detecting a problem is customer support: if users are complaining to the support team that something is broken or slow, well, that's one way to discover the problem. Automated monitoring maybe can't find everything every time, but we shouldn't be relying on notifications from the support team as the primary mechanism.

Suppose that we've detected a situation that we think needs some attention. Niall Murphy says there are three ways to respond:

- **Alerts**: a human must take action now;
- **Tickets**: a human must take action soon (hours or days);
- **Logging**: no need to look at this except for forensic/diagnostic purposes.

A common bad situation is logs-as-tickets: you should never be in the situation where you routinely have to look through logs to find errors. Write code to scan logs.

Alerting. Alerting is pretty much exactly what it sounds like: a person is notified about a problem. We usually talk about it in the metaphor of pagers, as in the small rectangular box that wakes up doctors when patients need them. Except now we're the "doctors" and the patients are software. So as the person who is on-call (or on-duty, or on-shift), when one of the conditions for alerting is fulfilled, you get notified, usually with some beeping.

The alert may contain some information about the event that has caused it, such as telling you that Service A CPU usage is high, but there's rarely enough in there to diagnose the problem. It's off to the dashboard, then.

It is very important to be judicious about the use of alerts. If your alerts are too common, they get ignored. When you hear the fire alarm in a building, chances are your thought is not "the building is on fire; I should leave it immediately in an orderly fashion.". More likely your reaction is "great, some jerk has pulled the fire alarm for a stupid prank or to get out of failing a midterm." This is because we have experienced too many false alarms, so we likely think that any alarm is a false one. It's a good heuristic; you'll be correct most of the time. But if there is an actual fire, you will not only be wrong, you might also be dead.

Still, alerts and tickets are a great way to make user pain into developer pain. Being woken up in the middle of the night (... day? A lot of programmers are nocturnal, now that I think of it) because of some SUPER CRITICAL notification that says OMG KITTENS ARE ENDANGERED is an excellent way to learn the lesson that production code needs to be written carefully, reviewed, QA'd, and perhaps run by a customer or two before it gets deployed to everyone. Developers, being human, will probably take steps to avoid their pain[73]. and they will take steps that keep these things from happening in the future: good processes and monitoring and all that goes with it.

[73] There is a great quotation to this effect by Frédéric Bastiat about how men will avoid pain and work is pain.

Incident Reports. Another important aspect of responding to an incident is an incident report, or post-mortem. This is an opportunity to try to identify the root causes of the issue and anything that we could learn from the situation. If we don't find the root cause, it could happen again. And if we don't learn anything, it's bad for the company (the same or similar problems will recur), and also bad for your own career growth. Therefore a report is worth doing, even if it's not fun.

The first thing that an incident report should contain is a breakdown of what happened, preferably with a clear timeline of events. The purpose of this is to understand when the problem started, when it was noticed, what actions were taken, and when the incident was resolved. Strictly the facts here, please. More on this later, but reports should not apportion blame. And it's not useful to blame someone when anyone else would have been likely to do the same thing.

The most important part is analysis of why the incident happened. There are two kinds of causes, root causes and proximate causes. Proximate causes are the things that happen just before the problem or are the trigger that makes it happen. If you remember talking about deadlock, a deadlock can occur because thread A acquires lock X then waits for lock Y, while thread B locks Y and then waits for X. The proximate cause is the lock acquisitions of threads A and B, such as because User 1 tried to update a record while User 2 was trying to run a report. That's not the root cause, though. The root cause is actually we don't always acquire locks in a consistent order.

It's key to focus on finding the root cause rather than stop at a superficial level. If we say the root cause is just "I wrote a bug" the solutions to that are just... don't do that, or write more tests. Those are part of the solution but they aren't going far enough, because we also need to think about the root causes. Perhaps the test environment is unrepresentative of the production environment, so it's hard to assess the actual impact of a change before it goes live. Or the root cause could be a lack of knowledge about a particular subject (e.g., concurrency).

Toyota (yes, the car company) uses a model of "Five Whys", which is to say you should ask "why?" five times to get to the actual root cause. I don't think it's necessary to always use exactly five, but it's a guide to making sure we're looking at the right level. If we don't ask the question enough then our answers are too superficial, but if we ask it too much then we end up with a root cause of "writing code is hard" and we end up debating whether computers were a mistake.

Then there are action items: what will we do in the short and long term to solve the problem and keep it from happening again? Things might be as simple as fixing the bug, but maybe monitoring/alerting need to be improved? Ideally we can even think of something that would prevent this kind of problem from happening again in the first place, such as introducing a testbench deployment stage that would catch performance problems before they go to production.

Action items, obviously, only help if you do them, so they have to be realistic. While you could say that you could prevent the race condition problem by rewriting all the company code in Rust at once, and that might even be factually correct, but it's also unlikely to happen.

And the final things are about what lessons we learned. This can be relatively straightforward—not every incident report results in head-exploding revelations—but there should be at least a few takeaways. Things we learned don't have to be negative learnings: perhaps we make a conclusion that the monitoring system is effective and catches the problem quickly.

Lastly, there are some things that should not be in a report. Irrelevant detail is the easiest to explain: long reports that contain things that distract from the point make it less likely that people read it or take action on it. Speculation should also be avoided because it's often wrong; now sometimes we may be forced to proceed without being totally certain about why or what happened. Educated guessing and theorizing are okay, but speculation is just guessing without any basis. Finally, again, the report should not contain blaming or shaming; these are counterproductive for learning and changes.

Why is blaming bad? If you know you're going to get in trouble for an incident, it incentivizes covering up incidents or denying the problem. It also incentivizes people to spend time shifting blame to others or arguing about who is really at fault. There are lots of articles out there about how blaming results in worse patient outcomes in the medical field as compared to a learning culture.

But performance problems or downtime are not the only things you need to be looking out for... There's also making sure that attackers aren't abusing your platform and systems...

Security, Report to the Bridge

Having an always-available service accessible over the internet makes security a very big concern. You can run some program with tons of security vulnerabilities offline and feel that the security problems can be managed (though you might still be wrong in interesting ways), but when it's online the risk is enormous. All kinds of vulnerabilities are a problem, but I'll call out two of them as being especially bad: code execution/injection and data leakage (information exposure).

Code execution is exactly what it sounds like: attackers run their code using your platform. They can mess with your data, send spam or harrassing e-mails to your customers, take services without paying for them, mine bitcoin at your expense, and many other things. Sometimes these are company-ending events. Actually, let's go on a quick digression about monitoring, with a real life example about detecting the very bad behaviour of crypto mining.

Abusing Free Services for Fun and Profit. Our source here is an article about how some security researchers found some malicious actors using build and deployment and other systems to mine cryptocurrency [Mor22].

The attack in question is about abusing the free (as in, no charge) CPU time and other resources that cloud providers offer. This is nice of them, and most use these resources for non-nefarious purposes, such as how the course notes for this course are auto-compiled using Github actions. No longer offering free resources like this would make it a little harder for attackers to abuse, but we also need to worry about a scenario where the attacker gets inside your network and can use the resources you are paying for to mine cryptocurrency for them. This costs money, yes, but also may slow down or hurt your actual operations since resources are limited, after all.

Part of the attack also involves bypassing normal signup limitations, including defeating CAPTCHAs and scripting the use of a web browser to make it look as though a human is browsing the page and clicking on the buttons. This suggests that malicious actors could swamp your services in fake/bot signups. Typically, there are some resources related to creating a new account, and if the malicious actor is creating accounts at a very fast pace, we might not be able to keep up, or at the very least every database will be full of extra data, so each insertion or retrieval becomes more expensive.

Once accounts are in hand, mining cryptocurrency in the way that's outlined in [Mor22] is not necessarily the most efficient route: given the number of accounts needed and the limitations of free tier resources, it's estimated that the attackers would need a few thousand free accounts to mine one coin. The return on investment is miserable from one point of view: using about $103 000 of resources produces one coin worth about $137, which is like a 0.13% return. Except, of course, it's Github who is paying the $103 000, so the return on paying nothing to get $137 is basically infinite. It's free money.

Of course, if someone looked at running jobs and saw a container called `mine-crypto-lol` deployed then it would certainly look suspicious, so the mining images and apps are given names to make them look much less conspicuous, e.g., `linux88884474` and `linuxapp84744474447444744474`. They also generate random GitHub action names to reduce the chance of detection there also. This makes it hard to detect, at a glance, rogue containers or processes.

Something like crypto mining may be easier to spot than other malicious activity, because it is CPU-intensive, so if we see some containers using a lot of CPU without an explanation or without knowing what the service does, we will likely feel compelled to investigate. But a container that's just sitting there quietly, observing things and sending your company's secrets to the competition or extortion rings? That's harder to find and could also be very costly.

Information Exposure. Information exposure is not only terrible for your company's reputation, but also against privacy laws and data protection regulations. A breach of the EU GDPR can get very expensive. See `https://www.enforcementtracker.com/` to find out which companies have recently gotten their wrists slapped or a huge fine. At the time of writing, the record holder is Amazon Europe (based in Luxembourg) with a fine of 746,000,000 EUR which is just over a billion CAD (using exchange rates from October 2022). Not exactly pocket change. Other big offenders are whom you might expect: Facebook (Meta), Google, but also H&M—yes, the clothing store—is a top 10 in this category.

You will notice that a lot of companies get fined for things like insufficient technical measures. That is to say, they don't do enough to avoid the data getting into the hands of the wrong people.

One situation I've seen that is potential nightmare in the making is when personally-identifiable-information (PII) is put into the logs. As long as the logs themselves remain secure, then there's no problem... but if an attacker can view the logs, they've got a one-stop-shop for everything they should not be able to see.

Why log these things in the first place? The security policies forbade the developers from accessing—even in read-only mode—the databases, so to debug and trace the code, excessive data was being put into the logs. So developers could see what they needed then. That's actually a fun example of what happens when security policies backfire, like when password policies are so restrictive that it just leads to users writing the password on sticky notes.

Defending against Vulnerabilities There are some companies out there that will check your code for libraries with versions having known security vulnerabilities. This is transitive, so if your app depends on some library that depends on a parsing library with a vulnerability, the vulnerability is noticed and reported. Each vulnerability is usually assigned a severity and the service may even suggest that updating from version X to Y will solve the problem. They can say this because they observe that a vulnerability reported in version X of a library is reported as fixed in version Y. Note, however, that just because there is an alert about a potential vulnerability does not mean that you actually use the vulnerable API. One of my (PL) students, along with a collaborator, showed that modularization improves the performance of these alerting tools; see [ADL24].

Sadly, when there is an updated version of a library, there may be breaking changes in it, so an upgrade might be more painful than just changing a version number and rebuilding. It may also take time for library authors to correct the vulnerability and release a fixed version of the library. In the meantime, you may need to implement some mitigation of your own, or just keep an eye open for when the patch is released. To support your wait for upstream, the vulnerability checks give you the ability to ignore or snooze the alert. Tempting as it is to do that and get on with other work, it's leaving the vulnerability in your code. Good practice would be for reviewers to discourage ignoring if it can be avoided.

As with the other parts of managing your application, like build and deployment, checking for vulnerabilities should be an automatic process as part of your build and release procedures.

Near-Miss? `xz` and `ssh`. Early in 2024, a vulnerability was discovered in the `xz` library and Andres Freund posted about it on the mailing list [Fre24]. The library itself is used for data compression but some versions of `openssh` rely on it. This is a strong example of what's termed a "supply chain attack".

The term comes from the logistical supply chain. Supply chains for physical products are often complicated; if you want to build a gaming PC you need a large number of different components and you only get the end product when you've got all the pieces at hand. The problem is recursive, though: how many different components and pieces did, for example, nvidia require to make your graphics card? If one of the components—however small or seemingly-insignificant—is unavailable for whatever reason, the device (or computer) can't be built.

Similarly, almost any modern piece of software that isn't a toy or academic assignment is built in a compositional way (app A uses libraries B, C, D each of which has dependencies $E, F, G, H...$), compromising any one of the dependencies F could result in a vulnerability in A. But if A is `openssh`, you know, the *secure* shell daemon, being able to exploit that means being able to have root access on the vulnerable server and execute arbitrary code.

Part of what makes this example so relevant for this course is that the problem was discovered via performance profiling! The backdoor makes the `openssh` server run much slower due to the injected code that is running before the authentication actually takes place; see the example from [Fre24]:

```
before:
nonexistant@...alhost: Permission denied (publickey).

before:
real 0m0.299s
user 0m0.202s
sys 0m0.006s

after:
nonexistant@...alhost: Permission denied (publickey).
```

```
real 0m0.807s
user 0m0.202s
sys 0m0.006s
```

The slides contain a graphic that gives a breakdown of the vulnerability in a (hopefully) concise way. There's a lot of obfuscation in the implementation and the conditions needed to trigger the vulnerability are very specific, to reduce the likelihood of discovery. The details of the vulnerability and the social engineering needed to land it in a library are better suited for a security course than this one—but this is an important lesson in how it's not enough to just defend against vulnerabilities in the libraries or tools we know we use.

Appendix A

A Crash Course on Threads

POSIX Threads

The term `pthread` refers to the POSIX standard (also known as the IEEE 1003.1c standard) that defines thread behaviour in UNIX and UNIX-like systems (Linux, Mac OS X, Solaris...). This is a specification document that says how threads should behave. This standard lets code for one UNIX-like system (e.g., Solaris) run easily on another (e.g., Linux). The POSIX standard for pthreads defines something like 100 function calls, but we need not examine all of them.

- `pthread_create` – Create a new thread.
- `pthread_exit` – Terminate the calling thread.
- `pthread_join` – Wait for a specific thread to exit. The caller cannot proceed until the thread it is waiting for calls `pthread_exit`. Note that it is an error to join a thread that has already been joined.
- `pthread_detach` – If we want to make it so that a thread cannot be joined, then we can make it a "detached" thread with this function.
- `pthread_yield` – Release the CPU and let another thread run. As they all belong to the same program, we expect that threads want to co-operate rather then compete for CPU time and threads can make decisions about when it would be ideal to let some other thread run instead.
- `pthread_attr_init` – Create and initialize a thread's attributes. The attributes contain things like the priority of the thread. ("After you, sir." "Oh no, after you.")
- `pthread_attr_destroy` – Remove a thread's attributes. Free up the memory holding the thread's attributes. This does not terminate the threads.
- `pthread_cancel` – Signal cancellation to a thread; this can be asynchronous or deferred, depending on the thread's attributes.
- `pthread_testcancel` – A thread can check to see if it has been cancelled. If that is the case, this function terminates the calling thread.

This list of functions gives us an overview of the toolkit we have, but we need to elaborate with some examples to fully understand how they work.

Creating a New Thread. When we want to start a new thread, we have to say what that new thread is supposed to do. The function signature for `pthread_create` looks like:

```
pthread_create( pthread_t *thread, const pthread_attr_t * attr, void *(*start_routine)( void * ), void *arg );
```

Where: `thread` is a pointer to a `pthread` identifier and will be assigned a value when the thread is created. The attributes `attr` may contain various characteristics (but you may supply `NULL` if you want the defaults). The third parameter is the function to run, but it requires a little more explanation.The last parameter, `arguments` is the argument passed to the `start_routine`. But that second last one is weird.

The `start_routine` parameter is the name of any function that takes a single untyped pointer and returns an untyped pointer. That is, the function signature has to match those two conditions. The name of the function (and the name of the argument) can be anything you like. See the example below:

```
void* do_something( void* start_params )
```

After the new thread has been created, the process has two threads in it. The OS makes no guarantee about which thread will be executing after the new one is created; this is a matter of scheduling. It could be either of the threads of the process, both of them at the same time, or a different process entirely.

Our experience with C-like languages suggests it is normal to have a single return value from a function, but usually we can have multiple input parameters. It seems limiting to be able to put in just one. There are two ways to get around this: with an array or with structures. In the case of the array, the argument provided to `pthread_create` is just a pointer to the array. This is also, incidentally, how you can get multiple return values out of a function in Java or C# (`public Object[] foo()`), but I don't recommend it as a good programming practice. The other way to do it is to use the `struct`, defining a structure for the parameter type and one for the return type.

The function that is to run in the new thread must expect a pointer to the arguments and then it will need to be cast to the appropriate (actual) type:

```
void* function( void * void_arg ) {
  parameters_t *arguments = (parameters_t*) args;
  /* continue after this */
}
```

This does imply that the caller of the `pthread_create` function has to know what kind of argument is expected in the function being called. That is fairly normal; we do have to know what the arguments mean when we pass them in to any function, but in this case we don't have the "hints" that the types provide.

What about the thread attributes? They can be used to set whether a thread is detached or joinable, scheduling policy, etc. By default, new threads are usually joinable (that is to say, that some other thread can call `pthread_join` on them). As noted before, it is a logical error to attempt multiple joins on the same thread. To prevent a thread from ever being joined, it can be created in the detached state (or the method `pthread_detach` can be called on a joinable thread). Trying to join a detached thread is also a logical error [Bar14]; testing tends to show that if you join with NULL you are ignored, but if you try to collect a value, your program will crash. For virtually all scenarios that we will consider in this course the default values will be fine.

Once we do that, the new thread we created is running. It does whatever its code does, so everything proceeds as expected, until of course the thread gets to the end. Usually, it will terminate with `pthread_exit`. The use of `pthread_exit` is not the only way that a thread may be terminated. Sometimes we want the thread to persist (hang around), but if we want to get a return value from the thread, then we need it to exit.

Returning Values. If a thread has no return values, it can just `return NULL;` which will have the same effect as `pthread_exit` and send `NULL` back to the thread that has joined it. If the function that is called as a task returns normally rather than calling the exit routine, the thread will still be terminated.

Another way a thread might terminate is if the `pthread_cancel` function is called with it as the target. As before, if the termination is deferred rather than asynchronous, the thread is responsible for cleaning up after itself before it stops.

A thread may also be terminated indirectly: if the entire process is terminated or if `main` finishes first (without calling `pthread_exit` itself). Indeed, `main` can use `pthread_exit` as the last thing that it does. Without that, `main` will not wait for other, unjoined threads to finish and they will all get suddenly terminated. If `main` calls `pthread_exit` then it will be blocked until the threads it has spawned have finished [Bar14].

Collecting Returned Values. Like the `wait` system call, the `pthread_join` is how we get a value out of the spawned thread:

```
pthread_join( pthread_t thread, void** retval );
```

The first parameter specifies the thread that you want to join. The second parameter is... wait... two stars? What we are looking for is a pointer to a void pointer. That is, we are going to supply a pointer that the join function will update to be pointing to the value returned by that function. Typically we supply the address of a pointer. This will be hopefully clearer in the example:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void * run( void * argument ) {
  char* a = (char*) argument;
  printf("Provided argument is %s!\n", a);
  int * return_val = malloc( sizeof( int ));
  *return_val = 99;
  pthread_exit( return_val );
}

int main( int argc, char** argv ) {
  if (argc != 2) {
      printf("Invalid args.\n");
      return -1;
  }
  pthread_t t;
  void* vr;

  pthread_create( &t, NULL, run, argv[1] );
  pthread_join( t, &vr );
  int* r = (int*) vr;
  printf("The other thread returned %d.\n", *r);
  free( vr );
  pthread_exit( 0 );
}
```

Thread Cancellation. Thread cancellation is exactly what it sounds like: a running thread will be terminated before it has finished its work. Once the user presses the cancel button on the file upload, we want to stop the upload task that was in progress. The thread that we are going to cancel is called the *target* (because we shoot targets, I guess) and there are two ways a thread might get cancelled [SGG13]:

1. **Asynchronous Cancellation:** One thread immediately terminates the target.
2. **Deferred Cancellation:** The target is informed that it is cancelled; the target is responsible for checking regularly if it is terminated, allowing it to clean itself up properly.

The pthread attributes can be used to set the cancellation type before it is created. A thread can declare its own cancellation type through the use of the function:

```
pthread_setcanceltype( int type, int *oldtype )
```

The first parameter is the new state we'd like this thread to take on, which would be one of the constants `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`. The second parameter will be updated to point to what the previous state was (although we might not care).

In deferred cancellation, a thread is responsible for checking if it has been cancelled, and if so, and stopping its activity and cleaning up (closing open files, etc.) before it terminates. It's possible, though generally poor programming practice (and very difficult), to never check for cancellation.

Given that a thread can effectively ignore a cancellation if it is the deferred cancellation type, why would we ever choose that over asynchronous cancellation? Suppose the thread we are cancelling has some resources. If the thread is terminated in a disorderly fashion, the operating system may not reclaim all resources from that thread.

Thus a resource may appear to be in use even though it is not, denying that resource to other threads and processes that may want to use it [SGG13].

The pthread command to cancel a thread is `pthread_cancel` and it takes one parameter (the thread identifier). By default, a pthread is set up for deferred cancellation. In the function that runs as a thread, to check if the thread has been cancelled, the function call is `pthread_testcancel` which takes no parameters.

Suppose your background task is to upload a bunch of files, consecutively. It is good programming practice to check `pthread_testcancel` at the start or end of each iteration of the loop, and if cancellation has been signalled, clean up open files and network connections, and then `pthread_exit`. Thus, if the thread has been told to cancel, it will do as it is told within a fairly short period of time.

It is noteworthy that a large number of functions are *cancellation points*; that is, the POSIX specification requires there is an implicit check for cancellation when calling one of those functions. There is an even larger number of functions that are "potential cancellation points", where the specification says that they could be cancellation points (but maybe aren't). You'll have to check the spec to see if that is the case for a specific function if there is a scenario where unexpected cancellation is a problem.

Now's not a good time! With the presence of cancellation points or asynchronous cancellation, sometimes a thread can be terminated before it has cleaned up some resources. This is undesirable. One way that we can guard against this is to register cleanup handlers for that thread. If, say, our thread allocated some memory, it would be wise to register a cleanup handler that deallocates that memory in case the thread should die unceremoniously. The function signatures are:

```
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */
```

To add a cleanup handler, the push function is used. Its two arguments are the function that is supposed to run, and a pointer to the argument that cleanup function will need.

The push function always needs to be paired with the pop function at the same level in your program (where level is defined by the curly braces). You should think of them as being like the opening curly brace at the start of a statement and the closing curly brace at the end; they have to be correctly matched up. The pop function takes one argument: whether it should run or not. If the thread is cancelled, the cleanup function will run; if it continues to the pop function, then you get to choose whether it runs or not.

Consider the following code:

```
void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_exit( NULL );
```

Suppose that the thread is cancelled during the block operating on `j` and it is set up for asynchronous cancellation. This means that the code will never get to the `free()` call, which means that the memory allocated at the beginning is leaked! We can remedy this with application of a cleanup handler:

```
void cleanup( void* mem ) {
  free( mem );
}

void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  pthread_cleanup_push( cleanup, j );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_cleanup_pop( 0 ); /* Don't run */
  pthread_exit( NULL );
```

Attributes and Using Memory to Pass Data. The earlier example used the return value of a thread. Sometimes, of course, we don't want to do that. One of the advantages of the use of threads is that data can be passed between

threads using memory directly. In this case, because there is no return value that we are about, we can use `NULL` in the call to join. This example also shows how to initialize and the attributes, although it doesn't override any of the defaults.

```
#include <pthread.h>
#include <stdio.h>

int sum; /* Shared Data */

void *runner(void *param);

int main( int argc, char **argv ) {

  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if ( argc != 2 ) {
    fprintf(stderr,"usage: %s <integer value>\n", argv[0]);
    return -1;
  }
  if ( atoi( argv[1] ) < 0 ) {
    fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
    return -1;
  }

  /* set the default attributes */
  pthread_attr_init( &attr );
  /* create the thread */
  pthread_create( &tid, &attr, runner, argv[1] );
  pthread_join( tid, NULL );
  printf( "sum = %d\n", sum );
  pthread_exit( NULL );
}

void *runner( void *param ) {
  int upper = atoi( param );
  sum = 0;
  for ( int i = 1; i <= upper; i++ ) {
    sum += i;
  }
  pthread_exit( 0 );
}
```

In this example, both threads are sharing the global variable `sum`. We have some form of co-ordination here because the parent thread will join the newly-spawned thread (i.e., wait until it is finished) before it tries to print out the value. If it did not join the spawned thread, the parent thread would print out the sum early.

Count to 10... Let's do a different take on that program:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum = 0;

void* runner( void *param ) {
    int upper = atoi( param );
    for (int i = 1; i <= upper; i++ ) {
        sum += i;
    }
    pthread_exit( 0 );
}

int main( int argc, char** argv ) {

    pthread_t tid[3];

    if ( argc != 2 ) {
        printf("An integer value is required as an argument.\n");
        return -1;
    }
    if ( atoi( argv[1]) < 0 ) {
        printf( "%d must be >= 0.\n", atoi(argv[1]) );
    }
```

```
    for ( int i = 0; i < 3; ++i ) {
        pthread_create( &tid[i], NULL, runner, argv[1] );
    }
    for ( int j = 0; j < 3; ++j ) {
        pthread_join( tid[j], NULL );
    }
    printf( "sum = %d.\n", sum );

    pthread_exit( 0 );
}
```

What's going wrong here? For very small values of the argument, nothing, but for a large number we get some strange and inconsistent results. Why? There are three threads that are modifying `sum`.

Summary: Relevant pthread Signatures

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **return_value )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* If the thread is cancelled, this function does not return (thread terminated) */
pthread_exit( void *value )
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */

sem_init( sem_t* semaphore, int shared, int initial_value); /* 0 for shared OK */
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```

Appendix B

A Review of Synchronization

Semaphore

A semaphore, outside of the context of computing, is a system of signals used for communication. Before ships had radios, when two friendly ships were in visual range, they would communicate with one another through flag semaphores, which is a fancy way of saying each ship had someone holding certain flags in a specific position. Thus the two ships could co-ordinate at a distance, even if the distance was limited to visual range. This worked dramatically better than many alternatives (e.g., shouting).

The computer semaphore was invented in 1965 by Edsger Dijkstra, a brilliant Dutch computer scientist who is sometimes maligned in textbooks as being eccentric or unusual. He described a data structure that can be used to solve synchronization problems via messages in [**?**]. Although the version we use now is not exactly the same as the original description, even 50+ years later, the core idea is unchanged.

We will begin with the *binary semaphore*: this is a variable that has two values, 0 and 1. It can be initialized to 0 or to 1. The semaphore has two operations: `wait` and `post`. In the original paper, `wait` was called `P` and `signal` was called `V`, but the names in common usage have become a little more descriptive. Mind you, if you can read/write Dutch as I can, the names make some sense: `P` is short for *proberen*, "to test", and `V` is short for *verhogen*, "to raise" or "to increment". But, for historical reasons as much as any other, the traditional lingua franca of computers is English, so the English names have tended to dominate. Furthermore, `post` is also called `signal` in many textbooks.

The `wait` operation on the semaphore is how a program tries to enter the critical section. When `wait` is called, if the semaphore value is 1, set it to 0 and this thread may enter the critical section and continue. If the semaphore is 0, some other thread is in the critical section and the current thread must *wait* its turn. The thread that called `wait` will be blocked by the operating system, just as if it asked for memory or a disk operation. This is sometimes referred to as decrementing the semaphore (because the value changes from 1 to 0).

The `post` operation is how a program sends the message that it is finished with the critical section. When this is called, if the semaphore is 1, do nothing; if the semaphore is 0 and there is a task blocked awaiting that semaphore, that task may be unblocked; else set the semaphore to 1. This is also sometimes called incrementing the semaphore.

If this is still confusing, consider the following analogy. Suppose you like coffee, and going to a particular coffee shop because there you can get your drink exactly the way you like it: half caf, no whip, extra hot, extra foam, two shot, soy milk latte[1]. After this delightful beverage it may be the case that you need to use the washroom. The washroom is locked at such places, so to get in you will need the key, which is available by asking one of the employees. If nobody is currently in the washroom, you will get the key and can proceed. If it is currently occupied, you will have to wait. When the key is returned, if anyone is waiting, the employee will give the key to the first person in line for the washroom; otherwise he or she will put the key away behind the counter.

[1] For the record, the author drinks tea, black.

Observe that the operating system is needed to make this work: if thread A attempts to wait on a semaphore that some other thread already has, it will be blocked and the operating system knows not to schedule it to run until it is unblocked. When thread B is finished and posts to the semaphore it is holding, that will unblock A and allow it to run again.

Note also that the semaphore does not provide any facility to "check" the current value. Thus a thread does not know in advance if it will block when it waits on the semaphore. It can only wait and may be blocked or may proceed directly into the critical section if there is no other thread in there at the moment.

When a thread signals a semaphore, it likewise does not know if any other thread(s) are waiting on that semaphore. There is no facility to check this, either. When thread A signals a semaphore, if another thread B is waiting, B will be unblocked and either thread A or thread B may continue execution (or both, if it is a SMP system), or another unrelated thread may be the one to continue execution. We have no way of knowing.

On the subject of observation, note that nothing about the semaphore as so defined protects against certain "bad" behaviour. Suppose thread C would like to enter the critical section. The programmer of this task is malicious as well as impatient: "my task is FAR too important to wait for those other processes and threads," he says, as he implements his code such that before he waits on the semaphore, he posts to it. Even though A or B might be in the critical section, the semaphore gets incremented so he is more or less certain that his program will now get to enter the critical section. It's not foolproof: if there are other threads waiting, they might get woken up to proceed instead of C; much depends on the scheduler. Nevertheless, this is really bad: one process can wreak all kinds of havoc by letting another process into the critical section. Though the example here makes the author of thread C a scheming villain (because the example is funnier that way), such a situation may occur without malicious intent if it is simply the result of a programming error.

The problem identified in the previous paragraph is usually solved by supplementing the basic binary semaphore. A data structure called a *mutex* (from **mut**ual **ex**clusion) is a binary semaphore with an additional rule enforced: only the thread that has called `wait` may `post` to that semaphore. This adds a small amount of extra bookkeeping to the semaphore, but this is a reasonable price to pay.

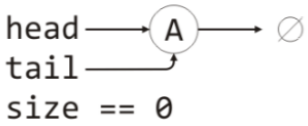
Example: Linked List Integrity

We will now examine a situation where a semaphore helps to prevent a synchronization error. This example comes from [HZMG15]. Imagine we have a shared linked list defined as:

```
typedef struct single_node {
  void *element;
  struct single_node *next;
} single_node_t;

typedef struct single_list {
  single_node_t *head;
  single_node_t *tail;
  int size;
} single_list_t;


void single_list_init( single_list_t *list ) {
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;
}

bool push_front( single_list_t *list, void *obj ) {
  single_node_t *tmp = malloc( sizeof( single_node_t ) );

  if ( tmp == NULL ) {
    return false;
  }

  tmp->element = obj;
  tmp->next = list->head;
  list->head = tmp;

  if ( list->size == 0 ) {
     list->tail = tmp;
```

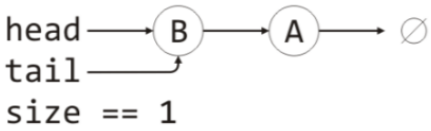
```
  }

   ++( list->size );
   return true;
}
```

If only one thread can access this data structure, we do not have a problem, but it was a shared linked list. Suppose a thread runs and tries to add an element A to the list using the `push_front` function. Right before the increment of the `size` field takes place there is a process switch. At this point, the new node has been allocated and initialized, the pointers of `head` and `tail` have been updated, but `size` is 0.



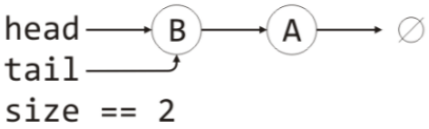
The linked list at the time of the thread switch [HZMG15].

Now, the second thread executes and wants to add B to the linked list. In the conditional statement, `list->size == 0` evaluates to true. Thus, the `tail` pointer is updated.



The linked list after the second thread adds B [HZMG15].

When the first thread gets to run again, it will resume where it left off: it increments the `size` integer, leaving the final state: `head` and `tail` both point to element B, even though there is element A in the list.



The linked list after the first thread resumes [HZMG15].

This is an *inconsistent state*: the linked list has two elements in it but the `tail` pointer is wrong. An attempt to remove an element from the list will reveal the problem, which can manifest in a few ways, depending on how the removal routine is implemented. If we try to remove the front element we might check that `head` and `tail` are equal, and that may give the mistaken impression that B is the last element in the list, so we "lose" A and it becomes a memory leak. Or perhaps the `head` pointer will be updated but `tail` will still point to B even after it has been freed, which can result in a segmentation fault or invalid access.

Semaphore Syntax

Binary semaphores are useful, and we can generalize this concept to what is known as a *counting* or *general* semaphore. Instead of having only the values 0 and 1, the setup routine for the counting semaphore allows the choice of an integer value and this is the maximum value. A thread that waits on that semaphore will decrement the integer by 1; a thread that signals on the semaphore will increment the integer by 1. If a thread attempts to wait on a semaphore and the decrement operation makes the integer value negative, the calling thread is blocked. If, however, the semaphore is, for example, initialized with 5 and the current value is 2, a thread that waits on that general semaphore will not be blocked.

In UNIX, the semaphores are always general. So, the functions are:

```
sem_init( sem_t* semaphore, int shared, int initial_value);
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```

Of these functions, the only one where the parameters are not obvious is the initialization routine. The parameter `shared` will be set to either 0 or 1: 1 if the semaphore is to be shared between processes (e.g., using shared memory), 0 otherwise. I'll also take a moment also to point out the importance of getting the initial value correct. If we choose the wrong initial value then our program might get stuck or we might not get the mutual exclusion behaviour we are supposed to have.

Applying the Semaphore to the Linked List

Now that we have the appropriate syntax we can apply it to the linked list example from this section. We will add to the linked list structure (`struct single_list`) a semaphore: `sem_t sem;`. In the initialization routine, we need to call the initialization method: `sem_init( &( list->sem ), 0, 1 );`

Finally, the `semaphore_wait` and `semaphore_signal` operations need to be added to `push_front` at the start and end of the critical section, respectively. Recall from earlier that we want the critical section to be as small as it can be. Putting it all together:

```
typedef struct single_node {
  void *element;
  struct single_node *next;
} single_node_t;

typedef struct single_list {
  single_node_t *head;
  single_node_t *tail;
  int size;
  sem_t sem;
} single_list_t;

void single_list_init( single_list_t *list ) {
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;

  sem_init( &( list->sem ), 0, 1 );
}

bool push_front( single_list_t *list, void *obj ) {
  single_node_t *tmp = malloc( sizeof( single_node_t ) );

  if ( tmp == NULL ) {
    return false;
  }

  tmp->element = obj;

  sem_wait( &( list->sem ) ); {
    tmp->next = list->head;
    list->head = tmp;

    if ( list->size == 0 ) {
       list->tail = tmp;
    }
    ++( list->size );

  } sem_post( &( list->sem ) );

  return true;
}
```

Strictly speaking, the braces ({ }) to enclose the critical region (between the wait and signal operations) are not necessary. This is just a use of C syntax to make it more obvious what the critical region is and to make it harder to make a mistake.

The critical section here just encloses the modification of the shared linked list. In theory one might put the wait

and signal operations at the start and end of the entire function, respectively. This is, however, suboptimal: it forces unnecessary waiting. In this specific example, including the call to `malloc` is especially bad; the memory allocation itself can block if insufficient memory is available. Thus the process currently in the critical section is blocked and that means no other thread can enter the critical section. This might result in the system getting totally stuck [HZMG15].

Mutex

```
int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 10000; ++i) {
        ++counter;
    }
}

int main(int argc, char *argv[]) {
    pthread_t threads[8];
    for ( int i = 0; i < 8; ++i ) {
      pthread_create( &threads[i], NULL, run, NULL );
    }
    for ( int j = 0; j < 8; ++j ) {
      pthread_join( threads[j], NULL );
    }
    printf("counter = %d\n", counter);
}
```

This produces different outputs on different runs of the program. Because there is a race condition, as we have seen before. We will use this pretty simple example to show how to use the mutex function calls.

Mutex Syntax

While it is possible, of course, to use a semaphore as a mutex, frequently we will use the more specialized tool for this task.

The structure representing the mutex is of type `pthread_mutex_t`. We don't care about the internals or what the struct is made of; it is either locked or unlocked and that's all that matters to us.

```
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )
```

The first function of note is `pthread_mutex_init` which is used to create a new mutex variable and returns it, with type `pthread_mutex_t`. It takes an optional parameter, the attributes (the details of which are not important at the moment, but relate mostly to priorities). We can initialize it using `NULL` and that is sufficient. There is also a syntactic shortcut to do static initialization if you do not want to set attributes [Bar14]:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

When created, by default, the mutex is unlocked. There are three methods related to using the mutex; two to lock it and one to unlock it, all of which take as a parameter the mutex to (un)lock. The unlock method, `pthread_mutex_unlock` is self-explanatory. As expected, attempting to unlock a mutex that is not currently locked is an error, but it is also an error if one thread attempts to unlock a mutex owned by another thread [Bar14].

The two kinds of lock are `pthread_mutex_lock`, which is blocking, and `pthread_mutex_trylock`, which is nonblocking. The lock function works as you would expect: if the mutex is currently locked, the calling function is blocked until its turn to enter the critical section; if the mutex is unlocked then it changes to being locked and the current thread enters the critical section. Trylock is more complicated and not necessary for understanding the producer-consumer example, but will come up again soon when we look at another classical synchronization problem.

To destroy a mutex, there is a method `pthread_mutex_destroy`. As expected, it cleans up a mutex and should be used when finished with it. If attributes were created with `pthread_mutexattr_init` they should be destroyed with `pthread_mutexattr_destroy`.

An attempt to destroy the mutex may fail if the mutex is currently locked. The specification says that destroying an unlocked mutex is okay, but attempting to destroy a locked one results in undefined behaviour. Undefined behaviour is, in the words of the internet, the worst thing ever: it means code might work some of the time or on some systems, but not others, or could work fine for a while and then break suddenly later when something else is changed[2].

So, let's apply that to the previous code:

```
pthread_mutex_t mutex;
int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        pthread_mutex_lock(&mutex);
        ++counter;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char *argv[]) {
    pthread_t threads[8];
    pthread_mutex_init( &mutex, NULL );
    for ( int i = 0; i < 8; ++i ) {
      pthread_create( &threads[i], NULL, run, NULL );
    }
    for ( int j = 0; j < 8; ++j ) {
      pthread_join( threads[j], NULL );
    }
    pthread_mutex_destroy(&mutex);
    printf("counter = %d\n", counter);
}
```

Later on we'll take a look into whether this code is optimal (it is not) and why, but at a first stage all that matters is correctness: preventing the race condition and ensuring consistent answers from the program. Correctness comes first, then performance...

Trylock In addition to locking the normal way, where we would get locked.

```
int pthread_mutex_trylock( pthread_mutex_t * mutex )
```

This function returns an integer and it's extremely important to check and see if the return code is 0, because that is the only way to know if the lock was acquired. The call is non-blocking so the code will carry on regardless. Consider below a code description of the dining philosophers if they used two-phase locking via the trylock routines. Assume that the mutex variables have been initialized appropriately. It should be possible to reason about this solution and demonstrate that (1) a philosopher can only eat if they have both chopsticks, and (2) deadlock does not occur.

```
int locked_both = 0;
while( locked_both == 0 ) {
  int locked1 = pthread_mutex_trylock( chopstick1 );
  int locked2 = pthread_mutex_trylock( chopstick2 );
  if (locked1 != 0 && locked2 == 0) {
    pthread_mutex_unlock( chopstick2 );
  } else if (locked1 == 0 && locked2 != 0 ) {
    pthread_mutex_unlock( chopstick1 );
  } else if (locked1 != 0 && locked2 != 0 ) {
    /* Do nothing */
  } else {
    locked_both = 1;
  }
}
```

[2]Sadly, the specifications for C and POSIX and many other things are riddled with these "undefined behaviour" situations and it causes programmers everywhere a great deal of stress and difficulty. Another example: reading from an uninitialized variable in C produces undefined behaviour too.

```
eat( );

pthread_mutex_unlock( chopstick1 );
pthread_mutex_unlock( chopstick2 );
```

Appendix C

Crossbeam and Rayon

Use Libs Wisely!

In previous courses, such as a concurrency course, there was a lot of expectation to do most things the hard way: write your own implementation and don't use libraries. In this course, such restrictions don't apply. In industry, you'll use libraries that have appropriate functionality, assuming the license for them is acceptable to your project. The two we'll talk about today, Crossbeam and Rayon are, for the record, Apache licensed, so it should pose no issue. In the previous version of the course where we used C and C++, we taught the OpenMP functionality, which is used to direct the compiler to parallelize things in a pretty concise way. The same idea applies here, except it's using the Crossbeam and Rayon crates rather than compiler directives.

Concurrency with Crossbeam

You'll recall from earlier when we introduced threads, that this doesn't work:

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's␣a␣vector:␣{:?}", v);
    });

    handle.join().unwrap();
}
```

The problem was that the compiler can't tell for sure how long the data is going to live, and we said that we can get around this by moving the vector into the thread and returning it if needed again. Then later, when we wanted to share a Mutex between threads, we used the `Arc` type. We could have used the `Arc` type here too, but that's a bit of a pain if the thread is going to be short-lived. Crossbeam gives us the ability to create "scoped" threads. Scope is like a little container we are going to put our threads in. It allows the compiler to be convinced that a thread will be joined before leaving the scope (container). Alright, how about this?

```
fn main() {
    let v = vec![1, 2, 3];

    crossbeam::scope(|scope| {
        println!("Here's␣a␣vector:␣{:?}", v);
    }).unwrap();

    println!("Vector␣v␣is␣back:␣{:?}", v);
}
```

Wait a minute. That's not quite right, because if I add statements where we print the thread ID, I get this output:

```
main thread has id 4583173568
Here's a vector: [1, 2, 3]
Now in thread with id 4583173568
Vector v is back: [1, 2, 3]
```

All we did was make the container, but we didn't spawn any threads. Here's a better version:

```
fn main() {
    let v = vec![1, 2, 3];
    println!("main␣thread␣has␣id␣{}", thread_id::get());

    crossbeam::scope(|scope| {
        scope.spawn(|inner_scope| {
            println!("Here's␣a␣vector:␣{:?}", v);
            println!("Now␣in␣thread␣with␣id␣{}", thread_id::get());
        });
    }).unwrap();

    println!("Vector␣v␣is␣back:␣{:?}", v);
}
```

With output:

```
main thread has id 4439997888
Here's a vector: [1, 2, 3]
Now in thread with id 123145430474752
Vector v is back: [1, 2, 3]
```

There are still rules, of course, and you cannot borrow the vector mutably into two threads in the same scope. This does, however, reduce the amount of ceremony required for passing the data back and forth. Wrapping everything in `Arc` is tedious, to be sure.

Producer-Consumer with channels. One of the guidelines that Rust gives is that it's preferable to do message passing as compared to shared memory. Unfortunately, for something like the producer-consumer scenario, the standard multiple-producer-single-consumer channel we are provided won't do; we need a multiple-producer-multiple-consumer channel. And sure enough, Crossbeam gives us one.

Recall the multi-producer multi-consumer example from earlier. We had to define a shared buffer structure and use semaphores and a mutex to coordinate access, and we saw it's possible to get it wrong in when we drop the mutex. Here's the excerpts from the Crossbeam version. First, a little setup to create a bounded channel with capacity the same as the bounded buffer (100):

```
let (send_end, receive_end) = bounded(CHANNEL_CAPACITY);
let send_end = Arc::new(send_end);
let receive_end = Arc::new(receive_end);
```

This uses the bounded channel, which has a maximum capacity as specified. If it's full, the sender is blocked until space becomes available. You can also choose an unbounded channel, which allows an arbitrary number at a time (but of course, they have to go somewhere so uncollected messages still take up space in memory and memory is not infinite, but close...). And look how much simpler the consumer is:

```
    for _j in 0 .. NUM_THREADS {
        // create consumers
        let receive_end = receive_end.clone();
        threads.push(
            thread::spawn(move || {
                for _k in 0 .. ITEMS_PER_THREAD {
                    let to_consume = receive_end.recv().unwrap();
                    consume_item(to_consume);
                }
            })
        );
    }
```

Certainly that's a lot simpler and cleaner, but is it faster? Testing says yes! Hyperfine says the original producer-consumer-opt version takes 372 ms to run for 10000 items consumed per thread, and the version with the channel takes 232.

Try Again Later. Another small thing that Crossbeam enables is an exponential backoff. When attempting to access some resource, we acknowledge that it might not be available right now. If that's the case, an error is not necessarily fatal and the client might want to retry. However, it's very unhelpful to have a tight loop that simply retries as fast as possible. What you should do instead is an exponential backoff: wait a little bit and try again, and if the error occurs, next time wait a little longer.

The idea is that if the resource is not available, repeatedly retrying doesn't help. If it's down for maintenance, it could be quite a while before it's back and calling the endpoint 5 or 10 times every second doesn't make it come back faster and just wastes effort. Or, if the resource is overloaded right now, the reaction of requesting it more will make it even more overloaded and makes the problem worse! And the more failures have occurred, the longer the wait, which gives the service a chance to recover.

Eventually, though, you may have to conclude that there's no point in further retries. At that point you can block the thread or return an error, but setting a cap on the maximum retry attempts is reasonable.

The `Backoff` util from Crossbeam gives you this functionality. Each step of the backoff takes about double the amount of time of the previous, up to a certain maximum.

Here's an example from the Crossbeam docs of using the backoff in a lock-free loop. Here, the `spin()` function is used because we can try again immediately. We can do so because if the compare-and-swap operation failed, it's because another did the compare-and-swap and got a chance to run.

```
use crossbeam_utils::Backoff;
use std::sync::atomic::AtomicUsize;
use std::sync::atomic::Ordering::SeqCst;

fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
    let backoff = Backoff::new();
    loop {
        let val = a.load(SeqCst);
        if a.compare_and_swap(val, val.wrapping_mul(b), SeqCst) == val {
            return val;
        }
        backoff.spin();
    }
}
```

If what we actually need is to wait for another thread to take its turn before we go, we don't want to spin, we want to "snooze". This means we'll be waiting longer for the thread to awaken.

```
fn spin_wait(ready: &AtomicBool) {
    let backoff = Backoff::new();
    while !ready.load(SeqCst) {
        backoff.snooze();
    }
}
```

In both cases, the `backoff` type has a function `is_completed` which returns true if the maximum backoff time has been reached and it's advised to give up. And an existing `backoff` can be re-used if it's reset with the unsurprisingly-named `reset` function.

Having read a little bit of the source code of the Crossbeam backoff, I'm not sure they implement a little randomness (called jitter). This is an improvement on the algorithm that prevents all threads or callers from retrying at the exact same time. Let me explain with an example: I once wrote a little program that tried synchronizing its threads via the database and had an exponential backoff if the thread in question did not successfully lock the item it wanted. I got a lot of warnings in the log about failing to lock, until I added a little randomness to the delay. It makes sense; if two threads fail at time X and they will both retry at time $X + 5$ then they will just fight over the same row. If one thread retries at $X + 9$ and another $X + 7$, they won't conflict.

The exponential backoff with jitter strategy is good for a scenario where you have lots of independent clients accessing the same resource. If you have one client accessing the resource lots of times, you might want something else; something resembling TCP congestion control. See [Aeo19] for details.

Data Parallelism with Rayon

Looking back at the nbody-bins-parallel code that we discussed earlier, you may have noticed that it contains some includes of a library called Rayon. It's a data parallelism library that's intended to make your sequential computation into a parallel one. In an ideal world, perhaps you've designed your application from the ground up to be easily parallelizable, or use multiple threads from the beginning. That might not be the situation you encounter in practice; you may instead be faced with a program that starts out as serial and you want to parallelize some sections that are slow (or lend themselves well to being done in parallel, at least) without a full or major rewrite.

That's what I wanted to do with the nbody problem. I was able to identify the critical loop (it is, unsurprisingly, in `calculate_forces`). We have a vector of points, and if there are N points we can calculate the force on each one independently.

My initial approach looked at spawning threads and moving stuff into the thread. This eventually ran up against the problem of trying to borrow the `accelerations` vector as mutable more than once. I have all these points in a collection and I'm never operating on one of them from more than one thread, but a compile time analysis of the borrowing semantics is that the vector is going to more than one thread. The compiler can't prove to itself that my slices will never overlap so they can't all be mutable.

This is a super common operation, and I know the operation I want to do is correct and won't have race conditions because each element in the vector is being modified only by the one thread. I eventually learned that you can split slices but it was going to be a slightly painful process. You can use `split_at_mut()` and it divides the slice into two pieces... it's a start, but would require doing this multiple times and probably use recursion or similar. Further research eventually told me to stop reinventing the wheel and use a library for this. Thus, Rayon.

Parallelizing loops. Back on track. A quick glance over this program tells us it is likely that the slow step is computing the interactions. It's reasonably common that computationally-intensive parts of the program happen in a loop, so parallelizing loops is likely to be quite profitable in terms of speeding things up. This is something that Rayon specializes in: it's easy to apply some parallelization directives to the loop to get a large speedup at a small cost (here, cost is programmer time in writing the change and reviewing it).

The line in question where we apply the Rayon library is:

```
accelerations.par_iter_mut().enumerate().for_each(|(i, current_accel)| {
```

A lot happens in this one line, so we need to take a look at it. Normally, we iterate over a collection (here, the accelerations) using an iterator (and it's preferable to the `for` construction). That's normally a sequential iterator, and the convenient thing about Rayon is that we can drop it in pretty easily. We need to include the "prelude". The prelude brings into scope a bunch of traits that are needed for the parallel iterators. Then instead of iterating over the collection sequentially, we use a parallel iterator that produces mutable reference. We effectively ask to have the slices cut up into slices of size 1. I also ask for the enumerate option because I'm going to use the index `i`, and then I provide the operation that I want to perform: a for-each (where I specify the index name and the name of the variable I want to use for each element in the collection.

Doing the work. The description we just saw of dividing up the work is how work units are specified, but doesn't cover what actually happens on execution. After all, if we just divide up the work without having more workers to do it, we're not getting anywhere.

The Rayon FAQ fills in some details about how work gets done. By default, the same number of threads are spawned as available (logical) CPUs (that means hypthereading cores count). These are your workers and the work is balanced using a work-stealing technique. If threads run out of work to do, they steal work from other queues. The technique is based off some foundational work in the area, called Cilk (see [FLR98] for details).

Maybe too easy? We discussed already the effectiveness of this change. And here's how easy it was:

```
diff live-coding/L14/nbody-bins/src/main.rs live-coding/L14/nbody-bins-parallel/src/main.rs
2a3
> use rayon::prelude::*;
64c65
```

```
<     for i in 0..NUM_POINTS {
---
>     accelerations.par_iter_mut().enumerate().for_each(|(i, current_accel)| {
66d66
<         let current_accel: &mut Acceleration = accelerations.get_mut(i).unwrap();
79,80c79
<     }
<
---
>     });
```

Why does the ease of doing this matter? Every change we introduce has the possibility of introducing a nonzero number of bugs. If I were to rewrite a lot of code, there would be more opportunities for bugs to appear. This change is minimal and easier for a reviewer to verify that it's correct than a big rearchitecting. And it's faster in terms of dev time – I can easily drop this in here and then move on to optimizing the next thing. Bugs slow you down.

It is important to note that the iterators do things in parallel, meaning the behaviour of the program can change a bit. If you are printing to the console or writing to a file or something, they can happen out of order when the loop is parallelized, just as they would if you wrote it so that different threads execute.

By that same token, the automatic parallelization doesn't mean there can be no race conditions; thus you still have to satisfy the compiler that your code is correct. If you are trying to find the largest item in an array, we still have to use atomic types or a mutex or similar to ensure that the correct answer is returned. See a quick example below:

```
use rayon::prelude::*;
use rand::Rng;
use std::i64::MIN;
use std::i64::MAX;
use std::sync::atomic::{AtomicI64, Ordering};

const VEC_SIZE: usize = 10000000;

fn main() {
    let vec = init_vector();
    let max = AtomicI64::new(MIN);
    vec.par_iter().for_each(|n| {
        loop {
            let old = max.load(Ordering::SeqCst);
            if *n <= old {
                break;
            }
            let returned = max.compare_and_swap(old, *n, Ordering::SeqCst);
            if returned == old {
                println!("Swapped {} for {}.", n, old);
                break;
            }
        }
    });
    println!("Max value in the array is {}", max.load(Ordering::SeqCst));
    if max.load(Ordering::SeqCst) == MAX {
        println!("This is the max value for an i64.")
    }
}

fn init_vector() -> Vec<i64> {
    let mut rng = rand::thread_rng();
    let mut vec = Vec::new();
    for _i in 0 ..VEC_SIZE {
        vec.push(rng.gen::<i64>())
    }
    vec
}
```

Here, the vector is iterated over in parallel and it's not using the mutable parallel iterator. This code example uses an atomic type, but if I change this to use a `Mutex` instead of an atomic type (see code below) it increases the runtime from about 121.6 ms to about 871.7 ms (tested with hyperfine as per usual).

```
fn main() {
    let vec = init_vector();
    let max = Mutex::new(MIN);
```

```
    vec.par_iter().for_each(|n| {
        let mut m = max.lock().unwrap();
        if *n > *m {
            *m = *n;
        }
    });
    let m = max.lock().unwrap();
    println!("Max value in the array is {}", m);
    if *m == MAX {
        println!("This is the max value for an i64.")
    }
}
```

The non-parallel code takes about 136.2 ms. This problem doesn't have enough work to parallelize effectively. While the lock-free version speeds it up a small amount, the mutex version was easier to write (and is therefore what you would probably do) but made it much slower. So it's important to consider carefully when to apply the library, because a speedup is not guaranteed just by parallelizing a given loop.

Bibliography

[Abb74] Abba. Waterloo, 1974. Online; accessed 14-December-2015. URL: https://www.youtube.com/watch?v=Sj_9CiNkkn4.

[ABuH+18] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. Cryptology ePrint Archive, Report 2018/1060, 2018. URL: https://eprint.iacr.org/2018/1060.

[ADL24] Mohammad Mahdi Abdollahpour, Jens Dietrich, and Patrick Lam. Enhancing security through modularization: A counterfactual analysis of vulnerability propagation and detection precision. In *Proceedings of the IEEE Conference on Source Code Analysis and Manipulation*, Flagstaff, AZ, USA, October 2024.

[Aeo19] Aeoncase. Improve on exponential backoff, 2019. Online; accessed 2020-10-21. URL: https://www.aeoncase.com/blog/posts/improve-on-exponential-backoff/.

[AKNJ21] Minwoo Ahn, Donghyun Kim, Taekeun Nam, and Jinkyu Jeong. Scoz: A system-wide causal profiler for multicore systems. *Software: Practice and Experience*, 51(5):1043–1058, 2021. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2930, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2930, doi:10.1002/spe.2930.

[AMP+20] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe Rust? In *Proceedings of the ACM on Programming Languages*, volume 4, November 2020. http://people.inf.ethz.ch/summersa/wiki/lib/exe/fetch.php?media=papers:unsafe-corpus.pdf.

[And15] Andre. Understanding Linux CPU load—when should you be worried?, 2015. Online; accessed 13-Feburary-2016. URL: http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages.

[Ast13a] Ankit Asthana. Building faster native applications, 2013. Online; accessed 8-January-2016. URL: https://blogs.msdn.microsoft.com/vcblog/2013/04/04/build-faster-and-high-performing-native-applications-using-pgo/.

[Ast13b] Ankit Asthana. Profile guided optimization, 2013. Online; accessed 8-January-2016. URL: http://nwcpp.org/talks/2013/ProfileGuidedOptimizationMarch21st.pptx.

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015. URL: https://computing.llnl.gov/tutorials/pthreads/.

[Bat23] James Batchelor. Unity clarifies new fee plans amid developer backlash, September 2023. Online; accessed 2023-10-14. URL: https://www.gamesindustry.biz/unity-clarifies-new-fee-plans-amid-developer-backlash.

[Can06] Bryan Cantrill. Hidden in Plain Sight, 2006. Online; accessed 20-Janaury-2016. URL: http://queue.acm.org/detail.cfm?id=1117401.

[Car24] Michael Carducci. The rise and fall of software architecture, 2024. Online; accessed 2025-06-07. URL: https://sufficiently-advanced.technology/post/rise-and-fall-of-architecture.

[CB15] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815409.

[CG10] Cliff Click and Brian Goetz. A crash course in modern hardware, 2010. Online; accessed 27-December-2016. URL: https://www.infoq.com/presentations/click-crash-course-modern-hardware.

[Cha18] Doug Chamberlain. Containers vs. Virtual Machines (VMs): What's the Difference?, 2018. Online; accessed 2019-12-16. URL: https://blog.netapp.com/blogs/containers-vs-vms/.

[Cor05] Microsoft Corporation. How to Use a Thread Pool (C# Programming Guide), 2005. Online; accessed 15-November-2015. URL: http://msdn.microsoft.com/en-us/library/3dasc8as%28v=vs.80%29.aspx.

[Cor20] Nvidia Corporation. Cuda C++ Programming Guide, 2020. Online; accessed 2020-10-15. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL: http://arxiv.org/abs/1810.04805, arXiv:1810.04805.

[Dev15] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler, 2015. Online; accessed 25-November-2015. URL: http://valgrind.org/docs/manual/cg-manual.html.

[Die09] Diego Novillo. LinkTimeOptimization, 2009. Online; accessed 22-December-2017. URL: https://gcc.gnu.org/wiki/LinkTimeOptimization.

[DKM+12] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording microprocessor history. *Queue*, 10(4):10:10–10:27, April 2012. URL: http://doi.acm.org/10.1145/2181796.2181798.

[DPS10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stoustroup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs, 2010. Online; accessed 14-December-2015. URL: http://www.stroustrup.com/isorc2010.pdf.

[Duf06] Joe Duffy. Anti-convoy locks in Windows Server 2003 SP1 and Windows Vista, 2006. Online; accessed 5-December-2017. URL: http://joeduffyblog.com/2006/12/14/anticonvoy-locks-in-windows-server-2003-sp1-and-windows-vista/.

[Dur15] Jonathan Dursi. HPC is dying, and MPI is killing it, 2015. Online; accessed 6-January-2016. URL: http://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it/.

[EHS19] Alexis Engelke, David Hildenbrand, and Martin Schulz. Optimizing performance at runtime using binary rewriting. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[EM15] Julia Evans and Kamal Marhubi. Do you know how much your computer can do in a second?, 2015. Online; accessed 28-October-2015. URL: http://computers-are-fast.github.io/.

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[Ent08] Sony Computer Entertainment. Cell programming primer, 2008. Online; accessed 6-January-2016. URL: https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingPrimer.html.

[Fac23a] Hugging Face. Fine-tune a pretrained model (v. 4.33.0), September 2023. Online; accessed 2023-09-16. URL: https://huggingface.co/docs/transformers/main/training#prepare-a-dataset.

[Fac23b] Hugging Face. Methods and tools for efficient training on a single GPU (v. 4.33.0), September 2023. Online; accessed 2023-09-11. URL: https://huggingface.co/docs/transformers/perf_train_gpu_one.

[Fac23c] Hugging Face. Model Training Anatomy (v. 4.33.0), September 2023. Online; accessed 2023-09-13. URL: https://huggingface.co/docs/transformers/model_memory_anatomy.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998. doi:10.1145/277652.277725.

[Fre24] Andres Freund. backdoor in upstream xz/liblzma leading to ssh server compromise, 2024. Online; accessed 2024-07-14. URL: https://www.openwall.com/lists/oss-security/2024/03/29/4.

[GNU16] GNU Compiler Collection. An inline function is as fast as a macro, 2016. Online; accessed 6-January-2016. URL: https://gcc.gnu.org/onlinedocs/gcc/Inline.html.

[Gre21] Andy Greenberg. An absurdly basic bug let anyone grab all of parler's data, January 2021. Online; accessed 2023-10-14. URL: https://www.wired.com/story/parler-hack-data-public-posts-images-video/.

[Gru13] Clemens Gruber. libcurl multi interface example, 2013. Online; accessed 30-October-2018. URL: https://gist.github.com/clemensg/4960504.

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework, 2014. 11th USENIX Symposium on Operating Systems Design and Implementation. URL: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-gonzalez.pdf.

[Han12a] Christian Plesner Hansen. 0x5f3759df, 2012. Online; accessed 2019-11-06. URL: http://h14s.p5r.org/2012/09/0x5f3759df.html.

[Han12b] Christian Plesner Hansen. 0x5f3759df (appendix), 2012. Online; accessed 2019-11-06. URL: http://h14s.p5r.org/2012/09/0x5f3759df-appendix.html.

[HB13] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems*. Cambridge University Press, 2013.

[HMS+09] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT CSAIL, Cambridge, MA, September 2009.

[Hor18] Jann Horn. Reading privileged memory with a side-channel, January 2018. Online; accessed 10-January-2018. URL: https://googleprojectzero.blogspot.ca/2018/01/reading-privileged-memory-with-side.html.

[How20] Jesse Howarth. Why Discord is switching from Go to Rust, 2020. Online; accessed 2020-09-12. URL: https://discord.com/blog/why-discord-is-switching-from-go-to-rust.

[Hub14] Jan Hubička. Linktime optimization in GCC, part 1–brief history, 2014. Online; accessed 22-December-2017. URL: http://hubicka.blogspot.ca/2014/04/linktime-optimization-in-gcc-1-brief.html.

[Hub15] Jan Hubička. Link time and inter-procedural optimization improvements in GCC 5, 2015. Online; accessed 22-December-2017. URL: http://hubicka.blogspot.ca/2015/04/GCC5-IPA-LTO-news.html.

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[JeG14] JeGX. AMD Radeon and NVIDIA GeForce FP32/FP64 GFLOPS Table, 2014. Online; accessed 2024-02-04. URL: https://www.geeks3d.com/20140305/amd-radeon-and-nvidia-geforce-fp32-fp64-gflops-table-computing/.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018. arXiv:1801.01203.

[Khu14] Paul Khuong. Performance tuning writing an essay, 2014. Online; accessed 26-January-2016. URL: http://www.pvk.ca/Blog/2014/10/19/performance-optimisation-~-writing-an-essay/.

[KMRS88] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988. URL: http://dx.doi.org/10.1007/BF01762111.

[KNC20] Steve Klabnik, Carol Nichols, and Rust Community. The Rust Programming Language, 2020. Online; accessed 2020-09-12. URL: https://doc.rust-lang.org/book/title-page.html.

[Kre13] Yossi Kreinin. How profilers lie: the cases of gprof and KCachegrind, 2013. Online; accessed 26-January-2016. URL: http://yosefk.com/blog/how-profilers-lie-the-cases-of-gprof-and-kcachegrind.html.

[KSK+23] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, Stephan Krusche, Gitta Kutyniok, Tilman Michaeli, Claudia Nerdel, Jürgen Pfeffer, Oleksandra Poquet, Michael Sailer, Albrecht Schmidt, Tina Seidel, Matthias Stadler, Jochen Weller, Jochen Kuhn, and Gjergji Kasneci. ChatGPT for good? on opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103:102274, 2023. URL: https://www.sciencedirect.com/science/article/pii/S1041608023000195, doi:10.1016/j.lindif.2023.102274.

[KSNH15] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015. doi:10.1109/MS.2014.50.

[Kul09] Kestas Kuliukas. How rainbow tables work, 2009. Online; accessed 17-December-2018. URL: http://kestas.kuliukas.com/RainbowTables/.

[Kur20] Merrin Kurian. Why should anyone use Apache Kafka?, 2020. Online; accessed 2020-11-07. URL: https://medium.com/swlh/why-should-anyone-use-apache-kafka-f2b632d0963c.

[KVN+08] A. Kejariwal, A.V. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the Intel Core 2 Duo processor. In *Proceedings, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation; SAMOS*, 2008.

[Lem18] Daniel Lemire. Multicore versus SIMD instructions: the "fasta" case study, 2018. Online; accessed 03-January-2018. URL: https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/.

[Liu09] Henry H. Liu. *Software Performance and Scalability: A Quantitative Approach*. John Wiley & Sons, 2009.

[LKR18] Daniel Lemire, Nathan Kurz, and Christoph Rupp. Stream VByte: Faster byte-oriented integer compression. *Information Processing Letters*, 130(Supplement C):1 – 6, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0020019017301679.

[LLV17] LLVM Project. LLVM link time optimization: Design and implementation, 2017. Online; accessed 22-December-2017. URL: https://llvm.org/docs/LinkTimeOptimization.html.

[Loh05] Sue Loh. Lock Convoys and How to Recognize Them, 2005. Online; accessed 3-December-2017. URL: https://blogs.msdn.microsoft.com/sloh/2005/05/27/lock-convoys-and-how-to-recognize-them/.

[Lop16] Crista Videira Lopes. Laws of performant software, 2016. Online; accessed 28-December-2016. URL: http://tagide.com/blog/advice/laws-of-peformant-software/.

[Lov13] Robert Love. What is the ideal design for a server process in Linux that handles concurrent socket I/O, 2013. Online; accessed 23-November-2015. URL: https://plus.google.com/+RobertLove/posts/VPMT8ucAcFH.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018. arXiv:1801.01207.

[Luu16] Dan Luu. The Nyquist theorem and limitations of sampling profilers today, with glimpses of tracing tools from the future, 2016. Online; accessed 1-February-2016. URL: http://danluu.com/perf-tracing/.

[Luu17] Dan Luu. A history of branch prediction from 1500000 bc to 1995, 2017. Online; accessed 5-December-2017. URL: http://danluu.com/branch-prediction/.

[LVVLP15] Mario Linares-Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. How developers detect and fix performance bottlenecks in Android apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 352–361, 2015. doi:10.1109/ICSM.2015.7332486.

[Mas18] Jon Masters. What are Meltdown and Spectre? here's what you need to know, January 2018. Online; accessed 10-January-2018. URL: https://www.redhat.com/en/blog/what-are-meltdown-and-spectre-here%E2%80%99s-what-you-need-know.

[MB23] John Maeda and Matthew Bolaños. What are Models?, May 2023. Online; accessed 2023-09-10. URL: https://learn.microsoft.com/en-us/semantic-kernel/prompt-engineering/llm-models.

[McD15] G.L. McDowell. *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup, LLC, 2015. URL: https://books.google.com/books?id=jD8iswEACAAJ.

[McS15] Frank McSherry. Scalability! but at what COST?, 2015. Online; accessed 11-January-2016. URL: http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html.

[Mel21] Leandro Melendez. *The HitchHiking Guide to Load Testing Projects: A Fun, Step-By-Step Walk-Through Guide*. Journeyman Publishing LLC, 2021. URL: https://books.google.ca/books?id=_eWAzgEACAAJ.

[Men08] Gaetano Mendola. False sharing hits again!, 2008. Online; accessed 7-December-2018. URL: http://cpp-today.blogspot.com/2008/05/false-sharing-hits-again.html.

[Mic11] Paulius Micikevicius. Local memory and register spilling. https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf, 2011.

[Mor22] Crystal Morin. Sysdig TRT uncovers massive cryptomining operation leveraging GitHub Actions, October 2022. Online; accessed 2022-10-29. URL: https://sysdig.com/blog/massive-cryptomining-operation-github-actions/.

[N+20] Nicholas Nethercote et al. *The Rust Performance Book*. Self-published, 2020. Online; accessed 2020-11-24. URL: https://nnethercote.github.io/perf-book/.

[Nie93] Jakob Nielsen. Response times: The 3 important limits, January 1993. Online; accessed 5-December-2017. URL: https://www.nngroup.com/articles/response-times-3-important-limits/.

[O'C18] Robert O'Callahan. Diagnosing a weak memory ordering bug, 2018. Online; accessed 2020-10-09. URL: https://robert.ocallahan.org/2018/08/for-first-time-in-my-life-i-tracked.html.

[Ora10] Oracle. Class ThreadPoolExecutor, 2010. Online; accessed 15-November-2015. URL: http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html.

[Ost04] Larry Osterman. So you need a worker thread pool..., 2004. Online; accessed 4-December-2017. URL: https://blogs.msdn.microsoft.com/larryosterman/2004/03/29/so-you-need-a-worker-thread-pool/.

[Oza18] Brent Ozar. Common entity framework problems: N + 1, 2018. Online; accessed 2025-06-02. URL: https://www.brentozar.com/archive/2018/07/common-entity-framework-problems-n-1/.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009. Online; accessed 6-January-2016. URL: http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf.

[Per16] Adam Perry. Rust Performance: A story featuring perf and flamegraph on Linux, 2016. Online; accessed 2020-11-01. URL: https://blog.anp.lol/rust/2016/07/24/profiling-rust-perf-flamegraph/.

[Per21] Kevin Perjurer. Disney's FastPass: A Complicated History, November 2021. Online; accessed 2022-11-29. URL: https://www.youtube.com/watch?v=9yjZpBq1XBE.

[Pre12] Jeff Preshing. Weak vs. strong memory models, 2012. Online; accessed 2020-10-09. URL: https://preshing.com/20120930/weak-vs-strong-memory-models/.

[R+15] Eric Rowell et al. Know thy complexities!, 2015. Online; accessed 14-November-2015. URL: http://bigocheatsheet.com/.

[RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of Onward! 2010*, pages 806–821, Reno/Tahoe, NV, USA, October 2010. ACM. URL: http://doi.acm.org/10.1145/1932682.1869525.

[Rin07] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of OOPSLA 2007*, pages 369–386, Montreal, Quebec, Canada, October 2007.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sig09] Karl Sigman. Notes on Little's Law, 2009. Online; accessed 4-April-2018. URL: http://www.columbia.edu/~ks20/stochastic-I/stochastic-I-LL.pdf.

[Sit21] Richard L. Sites. *Understanding Software Dynamics*. Addison-Wesley Professional, 2021.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.

[Spo01] Joel Spolsky. Don't let architecture astronauts scare you, 2001. Online; accessed 2025-06-07. URL: https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/.

[Spo05] Joel Spolsky. The perils of JavaSchools, 2005. Online; accessed 8-December-2015. URL: http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory, 1995. Online; accessed 1-March-2019. URL: https://groups.csail.mit.edu/tds/papers/Shavit/ShavitTouitou-podc95.pdf.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Str18] Jakub Stransky. HotSpot JVM JIT optimisation techniques, 2018. Online; accessed 2020-11-13. URL: https://jakubstransky.com/2018/08/28/hotspot-jvm-jit-optimisation-techniques/.

[Tan05] Brian K. Tanaka. Monitoring Virtual Memory with vmstat, 2005. Online; accessed 13-February-2016. URL: http://www.linuxjournal.com/article/8178.

[Tay17] Michael Bedford Taylor. The evolution of Bitcoin hardware. *Computer*, 50(9):58–66, 2017.

[Tea06] Lighty Team. Lighty 1.5.0 and Linux-aio, 2006. Online; accessed 23-November-2015. URL: http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/.

[The15] The GNOME Project. Thread pools, 2015. Online; accessed 15-November-2015. URL: http://library.gnome.org/devel/glib/unstable/glib-Thread-Pools.html.

[Ton09] Tuomas Tonteri. A practical guide to SSE SIMD with c++, 2009. Online; accessed 2019-12-08. URL: http://sci.tuomastonteri.fi/programming/sse.

[Tur15] Aaron Turon. Lock-freedom without garbage collection, 2015. Online; accessed 2020-10-03. URL: https://aturon.github.io/blog/2015/08/27/epoch/#lock-free-data-structures.

[Whi98] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4, 10 1998. doi:10.1007/BF00175354.

[Wil10] Anthony Williams. Definitions of non-blocking, lock-free and wait-free, 2010. Online; accessed 9-December-2017. URL: https://www.justsoftwaresolutions.co.uk/threading/non_blocking_lock_free_and_wait_free.html.

[Wil13a] Ken Williams. COMP755 advanced operating systems: Calculating service times, 2013. Online; accessed 10-March-2016. URL: http://williams.comp.ncat.edu/comp755/CalculatingServiceTime.pdf.

[Wil13b] Ken Williams. COMP755 advanced operating systems: Queuing theory, 2013. Online; accessed 9-March-2016. URL: http://williams.comp.ncat.edu/comp755/Q.pdf.

[Wil13c] Ken Williams. COMP755 advanced operating systems: Transaction performance, 2013. Online; accessed 9-March-2016. URL: http://williams.comp.ncat.edu/comp755/PerfEvalSlidesQ.pdf.

[Wil20] Nick Wilcox. Target Feature vs Target CPU for Rust, July 2020. Online; accessed 2020-11-19. URL: https://www.nickwilcox.com/blog/target_cpu_vs_target_feature/.