Contents

1 — Programming for Performance 3

2 — Rust Basics 10

3 — Rust: Borrowing, Slices, Threads, Traits 17

4 — Rust: Breaking the Rules for Fun and Performance 24

5 — Asynchronous I/O 29

6 — Modern Processors 36

7 — CPU Hardware, Branch Prediction 43

8 — Cache Coherency 50

9 — Of Asgard & Hel 56

10 — Concurrency and Parallelism 59

11 — Use of Locks, Reentrancy 67

12 — Lock Convoys, Atomics, Lock-Freedom 73

13 — Dependencies and Speculation 80

14 — Early Termination, Reduced-Resource Computation 86

15 — Memory Consistency 91

16 — Rayon 95

17 — Mostly Data Parallelism 96

18 — Compiler Optimizations 101

19 — Optimizing the Compiler 108

20 — Performance Case Studies 109

21 — Laws of Performance & Performance Culture 116

22 — GPU Programming (CUDA) 121

23 — Password Cracking, Bitcoin Mining 129

24 — Profiling 133

25 — System-Level Profiling, Profiler Guided Optimization 141

26 — Liar, Liar 149

27 — Memory Profiling, Cachegrind 155

28 — Profiling and Scalability **162**
29 — Clusters & Cloud Computing **167**
30 — Introduction to Queueing Theory **171**
31 — Probability, Convergence, & Ergodicity **177**
32 — Applying Queueing Theory **182**
33 — Practical Scaling: Amazon AWS **187**
34 — DevOps for P4P **188**
35 — Rust **194**

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?

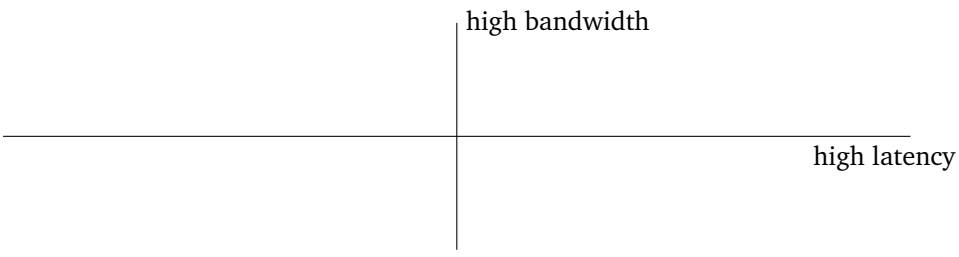
Let's think about the program execution as completion of some number of items—things to do. We have two concepts: items per unit time (bandwidth—more is better), and time per item (latency—less is better). Improving on either of these will make your program "faster" in some sense. In a way they are somewhat 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 [**?**].)

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 [**?**]. 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* [**?**]. 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.

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; 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.

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

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) 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. As I write this, it doesn't seem like Winter 2021 will be a good time to program together in a room with your non-roommate friends. 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, such as with MPI (and apply some queueing theory to find out how many servers you need!).

[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.

Acknowledgements

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

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 [**?**] 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 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 [**?**]).

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.

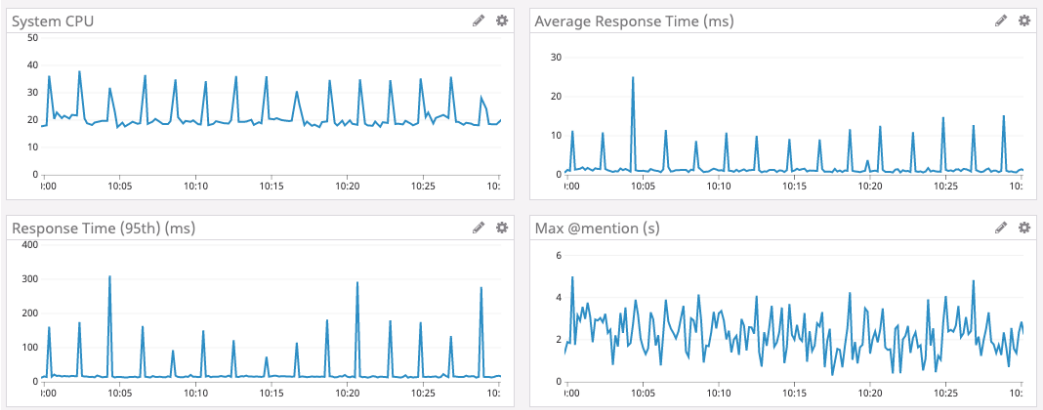
0wn3d

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 [**?**]:

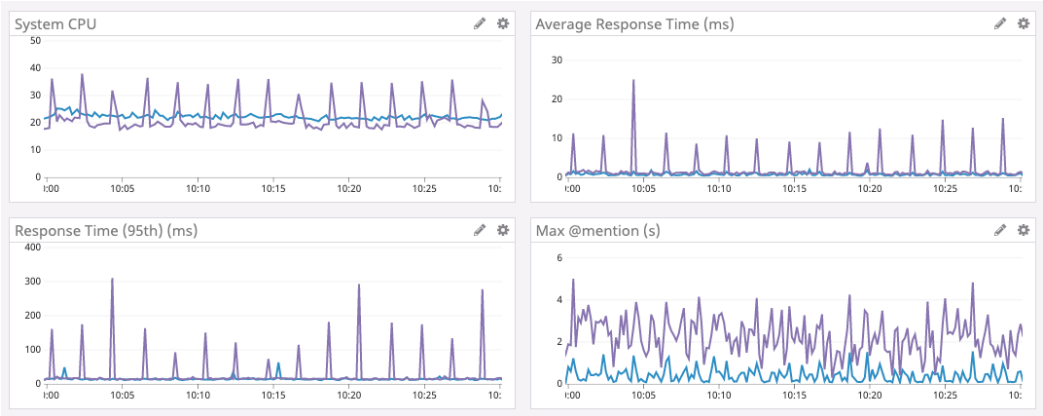


[6] `https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html`

[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`.

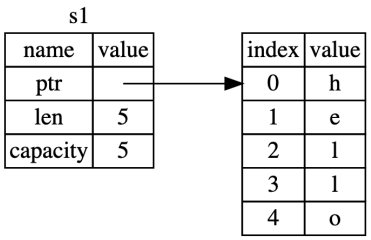
When `x` goes out of scope, then the memory will be deallocated ("dropped"). (This is very much like the RAII (Resource Acquisition Is Initialization) pattern in languages like C++). Variable scope rules look like scope rules in other C-like languages. We won't belabour the point by talking too much 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 [**?**]:



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.

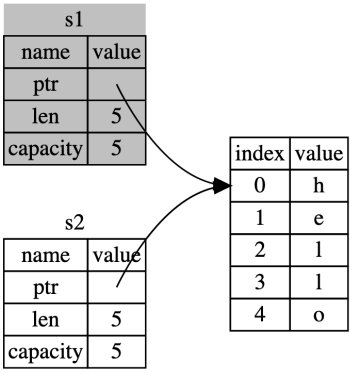
What's yours is mine. Move semantics have to do with transferring ownership from one variable to another. But ownership is overkill for simple types[8] (see the docs for a list—stuff like integers and booleans and floating point types), and such types don't need to follow move semantics; they follow 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 in that that when doing a pointer assignment like `thing* p = (thing*) ptr;` that 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 [**?**]:



[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) [**?**]:

```
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.

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.

For additional balance, 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.

[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.

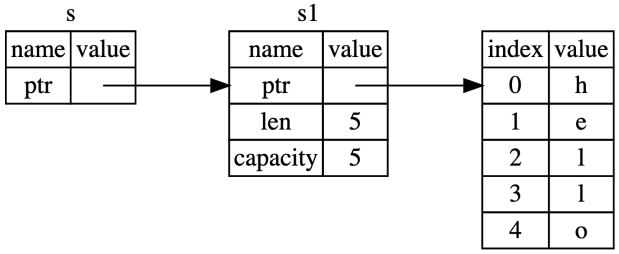
Rust's compiler analyzes all the borrowing that takes place in the program using the *borrow checker*. If the borrow checker is not certain that your code is perfectly safe, it will say no (and produce a compile time error). This can be a little bit frustrating, because the analysis is not perfect and 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 [**?**]:

```
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 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
}
```

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 and can eventually lead to 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;
}
```

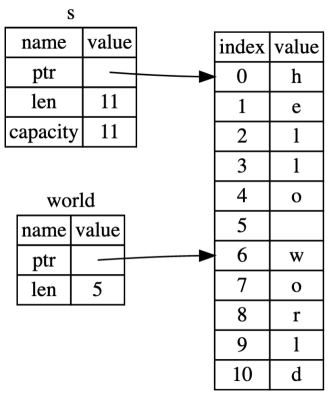
Under the old rules, the compiler would not allow creation of the mutable reference `z` because `y` has not gone out of scope. It would consider `y` to be valid until the end of the function. The improvement of NLL is that the compiler can see that `y` is no longer used after the `println!` macro and 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 [**?**]:



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 (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 explictly 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 enable you to 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. We will talk about the Rust way, but the background material will help you to be caught up.

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 [**?**]:

```
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);
    });

    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 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 Rust metaphor 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 [**?**]:

```
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 {
        fun 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 types of your own, 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. 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, 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, 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. And sample from the docs [**?**]:

```
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. 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 [**?**]:

```
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, after the name of the function, of our lifetime annotation 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, but fortunately the compiler can identify a lot of common scenarios, so the borrow checker can 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, like segmentation faults and memory leaks.

To do anything that qualifies as unsafe, you can go one of two ways. Either you declare a block as `unsafe`, or you specify that a given function is `unsafe` by putting that in the function signature. Inside an unsafe block or function, you can do the following things that you are not normally allowed to do [**?**]:

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.

The design intentions for unsafe blocks are that they are supposed to be small (this reduces the chance of an error and makes it easier to find) and ideally are abstracted away a bit behind some interface...

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();
}
```

Unsafe things are clearly designated in both ways: when you write a function that is unsafe, you declare to the world that this function is unsafe. Then, anyone who wants to use it also has to acknowledge that they know the function in question is unsafe. (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?

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. But if you find yourself doing such a thing, please stop and think very carefully about why.

Implement an unsafe trait. Appropriately, if the trait (interface) you want to implement has unsafe in the function signature, the compiler forces you to admit that your code is unsafe. If you do, it mostly means that you have to guarantee that what you're doing does in fact meet the requirements the interface specifies (like `Send`).

Unions. In C, there exists the concept of the `union`[10]. 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.

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 [**?**]:

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

```
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`). When you want 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. . .

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`[11] 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).

[11] `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; I don't know how to compile this from the command line. 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| {
    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 [**?**], 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[12]. 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 [**?**]:

- 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 undertand and easy to program."

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

Disadvantage:

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

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

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 [**?**].

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.

6 — Modern Processors

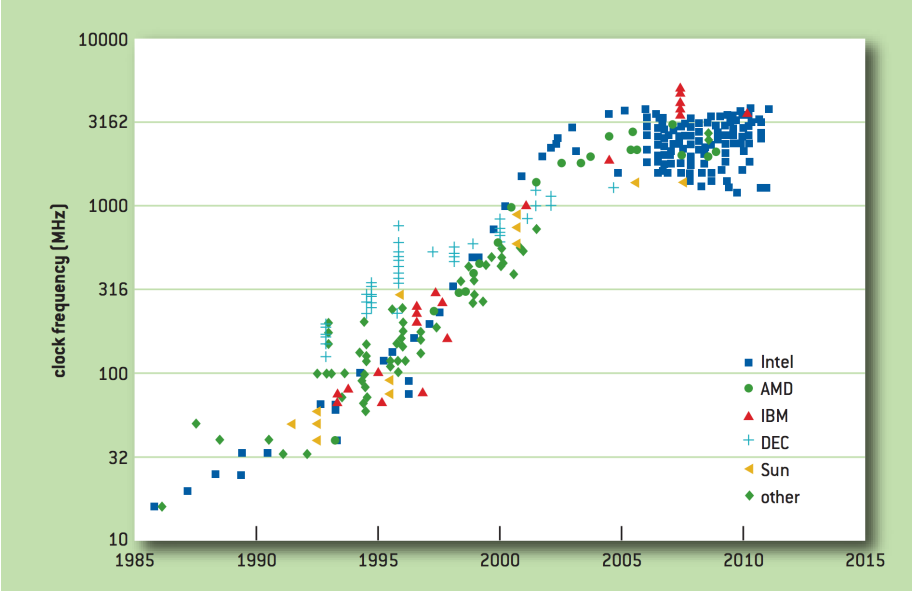
You know how `http://computers-are-fast.github.io/` featured in Lecture 1? It may also feature on your exams. You might want to print out you results and bring them.

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 [**?**].

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 [**?**]:



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 Computing (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)[13]. 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 Computing (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 [**?**] 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, 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[14]. 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.

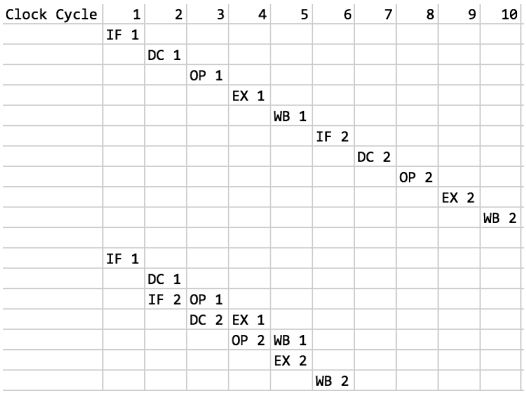
The final wall is the universal speed limit: the speed of light (curse you Einstein!). The more complex the CPU is, the longer the path any signal may have to travel to get from A to B. This is limited, most practically, by the speed of light, and thus far, nobody has invented a way to get around this 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:

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

[14] "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 put something in 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 [**?**]:

```
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)/ADD (read) 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.

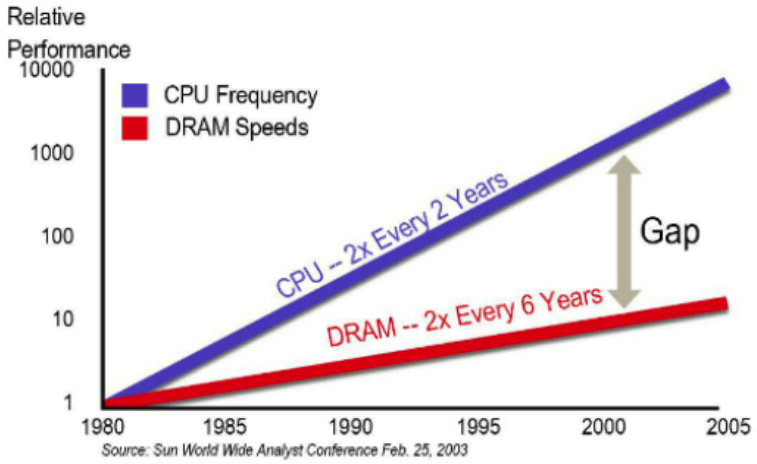
For years Intel was trying to push its Itanium processors (which were so unsuccessful they got the nickname "Itanic". Ouch). The goal of these was to find static (compile-time) parallelism: if a machine has infinite registers, can speculate infinitely, etc, the program gets sped up. Run all possibilities 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 an 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 [**?**] 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.

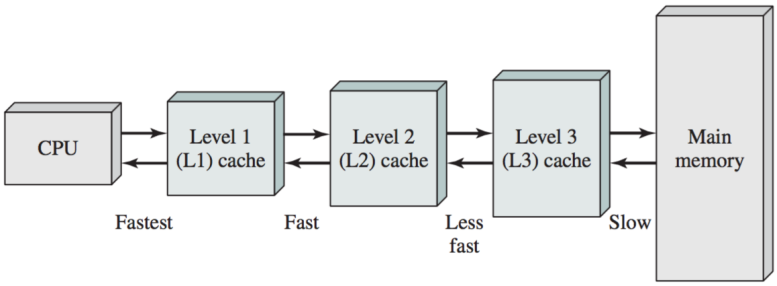
A Deeper Look at Cache Misses

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 [**?**].

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 [**?**].

Here are the reported cache miss rates[15] 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 formulae 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[16].

The slow step in all of this, is obviously, the amount of time it takes to load the page from disk. According to [**?**], 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.

[15] ‰ is "permil", or per-1000.

[16] 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.

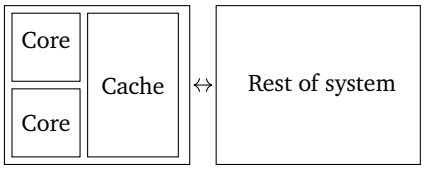
Chips and cores. Multiprocessor (usually SMP, or symmetric multiprocessor) systems have been around for a while. Such systems contain more than one CPU. We can count the number of CPUs by physically looking at the board; each CPU is a discrete physical thing.

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.

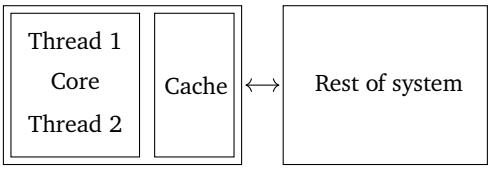
Hardware Designs for Multicores. In terms of the hardware design, cores might share a cache, as in this picture:



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

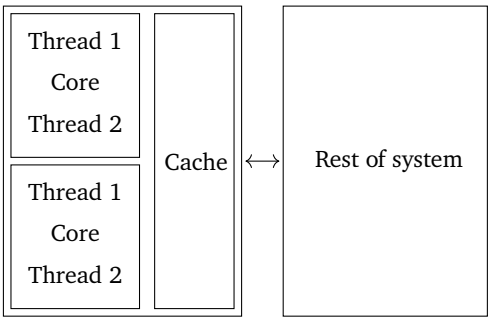
This above Symmetric Multithreading (SMP) design is especially good for the 1:1 threading model. In this case, the design of the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

Or, we can have a design that works well for the N:1 model:



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, then each thread would run more slowly. If they're doing different things, there's potential for speedup.

Finally, it's possible to both use multiple cores and put multiple threads onto one core, as in the M:N model:



Here we have four hardware threads; pairs of threads share hardware resources. One example of a processor which supports chip multi-threading (CMT) is the UltraSPARC T2, which has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

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.

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. In current hardware designs, each of the hardware threads has the same performance.

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.[17]

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`[18], 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

[17] Gove suggests that locality groups help reduce contention for core resources, but they seem to help more with memory.
[18] Which I still maintain is a swear word in C.

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

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 [**?**].

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

There's been a lot happening lately 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 [**?**] and Meltdown [**?**] 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 [**?**] by Jon Masters of RedHat. However, you should now have enough background to follow the Google Project Zero description at [**?**].

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

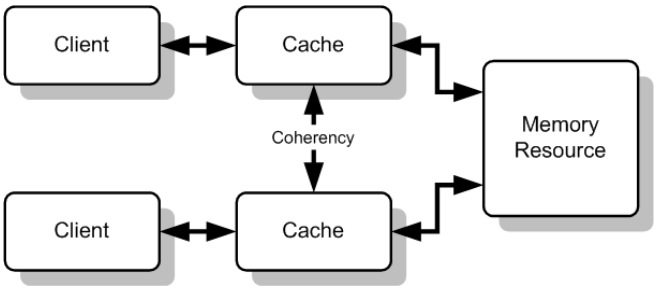
Multiprocessor (multicore) processors have some hardware that tries to keep the data consistent between different pipelines and caches (as we saw in the video). More processors, more threads means more work is necessary to keep these things in order. We will discuss cache coherence soon, but about hyperthreading...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



—Wikipedia

Today we'll look at what support the architecture provides for memory ordering, in particular in the form of cache coherence. Since this isn't an architecture course, we'll look at this material more from the point of view of a user, not an implementer.

The problem is, of course, that each CPU likely its own cache. If it does then these data 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:

- the values in all caches are consistent; and
- 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 chosen, we have to pick one. We can't pick none of those and expect to get the right answers.

Snoopy Caches. The simplest way to "do something" is to use Snoopy caches [**?**]. 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. [19] 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.

[19] 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.

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.

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`:
    - Generates a BusRdX.
    - `CPU1` snoops and invalidates x.
4. `CPU1` reads x:
    - 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

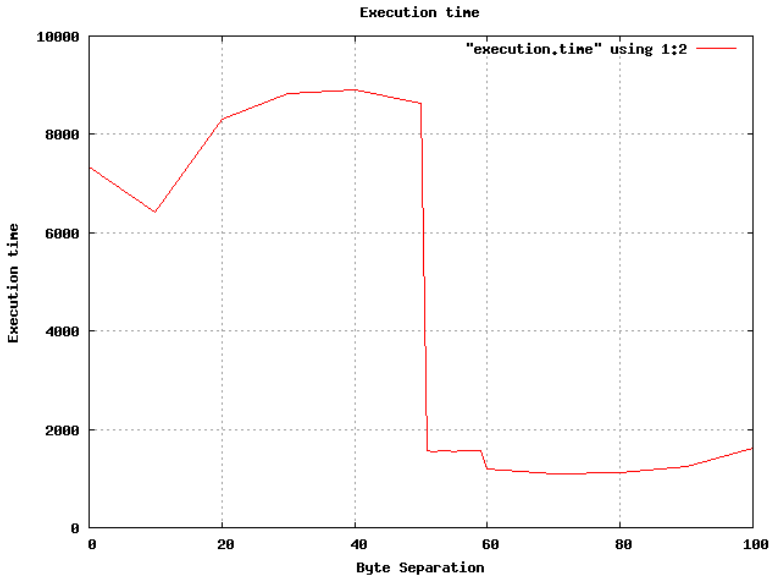
False sharing is something that happens when our program has two unrelated data elements that are mapped to the same cache line/location. Let's consider an example from [**?**]:

```
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, then `b` will be invalidated and the CPU working on `b` will be forced to fetch the newest value from memory. This can be avoided by seeing to it that there is some separation between these two arrays.

One way would be to heap allocate both arrays. You can find where things are located, if you are curious, by printing the pointer (or the address of a regular variable). Usually if you do this you will find that they are not both located at the same location. But you are provided no guarantee of that. 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 [**?**].

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!

P.S. putting these arrays in a struct and padding the struct can also help with enabling future updates to the struct.

9 — Of Asgard & Hel

Norse Mythology

Everything came into creation in the gap between fire and ice, and the World Tree (Yggdrasil) connects the nine worlds. Asgard is the home of the Æsir, the Norse gods. Helheim, or simply Hel, is the underworld where the dead go upon their death. In Hel or Asgard (it's not entirely clear), there is Valhalla, hall of the honoured dead. Those who die in battle and are judged worthy will be carried to Valhalla by the Valkyries. There they will reside until they are called upon to aid in Odin's fight with the wolf Fenrir in Ragnarök[20], the doom of the gods[21]. For the curious, humans live in the "middle realm", Midgård, surrounded by the serpent Jormungand, who will fight against Thor in Ragnarök. Thor will kill the serpent, but the serpent's poison will also finish off Thor[22].

Aside from my obvious passion about the subject, why are we talking about Norse Mythology? We're going to examine some very useful tools for programming called Valgrind and Helgrind (also Cachegrind). Note that the -grind endings on those are pronounced like "grinned". Where do they take their names from? Valgrind is the gateway to Valhalla; a gate that only the worthy can pass. Helgrind is the gateway to, well, Hel. Which despite being the source of the English word "Hell", is not the place where sinners go. It's just the place where the dead go.

But all of these, in program form, are analysis tools for your (usually) C and C++programs. They are absolute murder on performance, but they are wonderful for finding errors in your program. To use them you will start the tool of your choice and instruct it to invoke your program. 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. It's important to enable debugging symbols in your compile (`-g` option if using `gcc`) if you want stack traces to be useful.

This lecture is unlike the others in this course, in that it mostly revolves around in-class demonstrations. So if you missed it, well, you will need to play around with Valgrind/Helgrind yourself to see what it does. Fortunately, that is the next in-class exercise!

Valgrind (or Memcheck)

Valgrind is the base name of the project and by default it runs the memcheck tool. The purpose of memcheck is to look into all memory reads, writes, and to intercept and analyze every call to `malloc/free` and `new/delete`. Thus, memcheck will check all memory accesses and allocations/deallocations, and can find problems like:

- Accessing uninitialized memory
- Reading off the end of an array
- Memory leaks (failing to free allocated memory)
- Incorrect freeing of memory (double free calls or a mismatch)
- Incorrect use of C standard functions like `memcpy`

[20] German: Götterdämmerung - "Twilight of the gods"
[21] Spoiler alert: this isn't going to end well for Odin.
[22] Sorry if I've just spoiled the plot of a Marvel movie.

- Using memory after it's been freed.
- Asking for an invalid number of bytes in an allocation (negative?)

These errors will be reported to the console when they occur. Ideally, this will help you find the source of the problem. Ideally, you're going to see this:

```
==8476== All heap blocks were freed -- no leaks are possible
```

Okay, everything going perfectly is unlikely in anything other than a small program. If you take the program's suggestion to use `--leak-check=full` then you end up with a bit more detail about where memory was allocated. It can't tell you where the call to `free` should go, only where the memory that isn't freed was allocated.

From the Valgrind FAQ, how to read the leak summary:

- **Definitely lost**: a clear memory leak. Fix it.
- **Indirectly lost**: a problem with a pointer based structure (e.g., you've lost the head of the linked list, but the rest of the list is indirectly lost.) Generally, fixing the definitely lost items should be enough to clear up the indirectly lost stuff.
- **Possibly lost**: the program is leaking memory unless weird things are going on with pointers where you're pointing them to the middle of an allocated block.
- **Still reachable**: this is memory that was still allocated that might otherwise have been freed, but references to it exist so it at least wasn't lost.
- **Suppressed**: you can configure the tool to ignore things and those will appear in the suppressed category.

Still, it's also important to learn what to ignore (or what's out of our hands). The stack trace that we see will point us at the cause of the problem. Sometimes, though, we end up with something where there's very little involvement of our own program: it's thread creation or a library or similar. What do we do?

Well—we have to consider carefully if there's anything we can do about it. In a library, there might be an associated cleanup call that we have forgotten to use. Or maybe not, and the problem is beyond our ability to fix. You will have to consider carefully and use your judgement! Sorry. I know it's much better when there's a clear yes or no, but software is complex.

We'll take some time to do some examples that show the kind of error that Valgrind reports; both those we can do something with and those that are outside our control.

Helgrind

The purpose of Helgrind is to detect errors in the use of POSIX pthreads. In a way, Helgrind is a pretty neat tool for improving performance, even though it doesn't actually directly speed anything up. When we take a single-threaded program and split it off into a multithreaded program, we may introduce a lot of errors (or at least, introduce the possibility of a lot of errors). Truthfully, humans are not very good at parallel thinking; we are very much sequential. But a program that is fast and wrong is probably less useful than one that is slow and correct. Can we make it faster and still have it be correct? That's the goal of Helgrind: after you parallelize your code, it will do some automatic checking of the code to determine where, if anywhere, there are concurrency problems. It can't prove that your program is correct (if only) but it can at least catch some of the common problems you might introduce when writing a parallel program. Helgrind classifies errors into three basic categories:

1. Misuses of the pthreads API;
2. Lock ordering problems; and
3. Data races.

The first category does not require much explanation. These are just some programming errors related to the pthread API calls. Some examples from [**?**]:

- Unlocking a mutex that is unlocked;
- Deallocation of memory with a locked mutex in it; or
- Thread exit while holding a lock.

. . . and many more.

The second category of errors should be familiar to you from earlier as a source of potential deadlock.

Thread P

```
1. wait( a )
2. wait( b )
3. [critical section]
4. signal( a )
5. signal( b )
```

Thread Q

```
1. wait( b )
2. wait( a )
3. [critical section]
4. signal( b )
5. signal( a )
```

In this case, if the interleaving of these happens to work out in a couple of particular ways, then we get deadlock because thread P holds mutex `a` and thread Q holds mutex `b` and each waits for the mutex that the other one has. The example is slightly silly, of course, because it's super easy to see.

Helgrind builds a directed graph of lock acquisitions. When a thread acquires a lock, Helgrind checks to see whether a cycle exists. If so, then there is potential for a deadlock [**?**]. Helgrind will report as an error the initial order (the first order seen is the one viewed as "correct") and the "incorrect" order that is the source of the potential problem. Really, though, all that matters is consistency—following the same order. You may change either of the acquisition orders to match the other.

How does Helgrind work? It examines the use of the standard threading primitives—lock, unlock, signal/post, wait, etc. Anything that implies there might be an ordering between events is taken and added to a directed acyclic graph that represents these dependencies. If memory is accessed from two different threads and there is no path through this directed acyclic graph that indicates an ordering, then Helgrind reports a race [**?**]. Obviously, at least one of these accesses must be a write. (Recall: there is no read after read dependency).

Also cool: you can ask Helgrind to try to tell you about variable names (if it can) with the command line option `-read-var-info=yes`. Then it will tell you something interesting like:

```
==10454== Location 0x60104c is 0 bytes inside global var "var"
==10454== declared at datarace.c:3
```

These will give you indications of where you need to introduce synchronization of some kind (semaphore, mutex, condition variable, etc). The authors of Helgrind assume that if it tells you where the problem is, you will figure out what variables are affected and how to properly prevent data races. You might find this frustrating, in the sense of a serial complainer who thinks that he or she can just moan about what's wrong without bringing forward any suggestions about how to fix the problems.

We'll again take some time to examine some examples of each category of problem (as time allows).

10 — Concurrency and 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`.

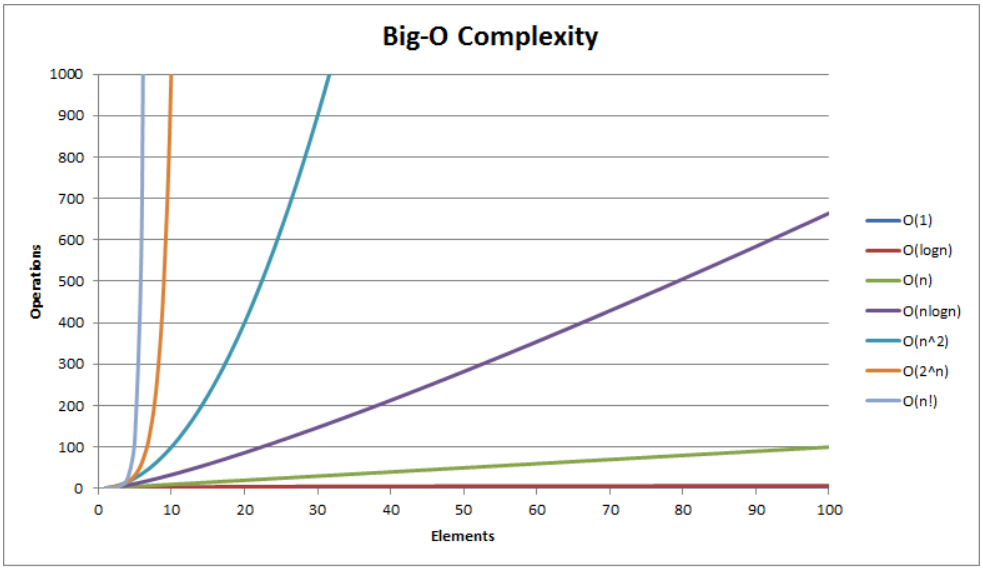
Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, 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.

Scalable 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 4^{th} year engineering course to have me tell you that you can use quicksort rather than bubble sort.



Big-O Complexity comparison from [?]

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

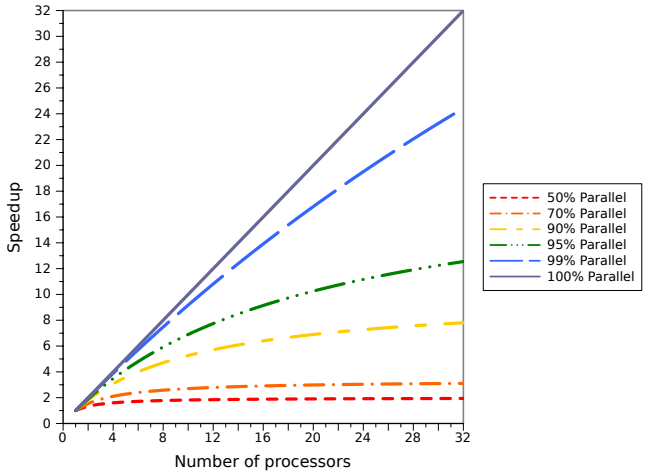
$$\text{max speedup} = \frac{1}{(1 - P)},$$

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 [**?**]: 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 last time, 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[23] 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 and 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 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.

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

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

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` [**?**], C#'s `System.Threading.ThreadPool` [**?**], and GLib's `GThreadPool` [**?**] all implement thread pools. There's a Rust crate called `thread pool`. 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.

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

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 i 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.
```

Threads and CPUs. In your operating systems class, you've seen implementations of threads ("lightweight processes"). We'll call these threads *software threads*, and we'll program with them throughout the class. Each software thread corresponds to a stream of instructions that the processor executes. On a old-school single-core, single-processor machine, the operating system multiplexes the CPU resources to execute multiple threads concurrently; however, only one thread runs at a time on the single CPU.

On the other hand, a modern chip contains a number of *hardware threads*, which correspond to the virtual CPUs. These are sometimes known as *strands*. The operating system still needs to multiplex the software threads onto the hardware threads, but now has more than one hardware thread to schedule work onto.

Choose Your Pain

The first design decision that you need to solve when parallelizing programs is whether you should use threads or processes. Threads are basically light-weight processes which piggy-back on processes' address space.

When processes are better. Processes are safer and more secure than threads.

1. Each process has its own virtual address space:
    - Memory pages are not copied, they are copy-on-write (usually in UNIX, anyway). Therefore, processes use less memory than you would expect.
2. Buffer overruns or other security holes do not expose other processes.
3. If a process crashes, the others can continue.

Example: In the Chrome browser, each tab is a separate process. Scott McCloud explained this: `http://www.scottmccloud.com/googlechrome/`.

When threads are better. Threads are easier and faster.

1. Interprocess communication (IPC) is more complicated and slower than interthread communication; must use operating system utilities (pipes, semaphores, shared memory, etc) which have system call overhead, instead of Rust communication's mechanisms (shared memory or message passing)
2. Processes have much higher startup, shutdown, and synchronization costs than threads.

How to choose? If your application is like this:

- mostly independent tasks, with little or no communication;
- task startup and shutdown costs are negligible compared to overall runtime; and
- want to be safer against bugs and security holes,

then processes are the way to go. If it's the opposite of this, then use threads.

For performance reasons, along with ease and consistency across systems, we'll use threads.

Overhead. The common wisdom is that processes are expensive, threads are cheap. Let's try it.

```
use std::process::Command;

fn main() {
    for j in 0 .. 50000 {
       Command::new("/bin/false").spawn();
    }
}
```

```
1.530 s +/-  0.134 s
```

```
use std::thread;

fn main() {
    for j in 0 .. 50000 {
        thread::spawn(|| {
            false
        });
    }
}
```

```
630.5 ms +/-  21.5 ms
```

Parallelization using Threads or Processes

We'll be looking at thread-based or process-based parallelization for the next bit. We don't care about the distinction between threads and processes for the moment. In fact, we could even distribute work over multiple systems. But the idea is, if you are looking to parallelize your program, you should think about which of the following patterns makes the most sense.

Pattern 1: Multiple Independent Tasks. If you're just trying to maximize system utilization, you can use one system to run a number of independent tasks; for instance, you can put both a web server and database on one machine. If the web server happens to be memory-bound while the database is I/O-bound, then both can use system resources. If the web server isn't talking to the database (rare these days!), then the tasks would not get in each others' way.

Most services probably ought to be run under virtualization these days, unless they're trivial or not mission-critical.

A more relevant example of multiple independent tasks occurs in cluster/grid/cloud computing: the cloud might run a number of independent tasks, and each node would run some of the tasks. The cloud can retry a task (on a different node, perhaps) if it fails on some node. Note that the performance ought to increase linearly with the number of threads, since there shouldn't be communication between the tasks.

Pattern 2: Multiple Loosely-Coupled Tasks. Some applications contain tasks which aren't quite independent (so there is some inter-task communication), but not much. In this case, the tasks may be different from each other. The communication might be from the tasks to a controller or status monitor; it would usually be asynchronous or be limited to exceptional situations.

Refactoring an application this way can help with latency: if you split off the CPU-intensive computations into a sub-thread, then the main thread can respond to user input more quickly.

Here's an example. Assume that an application needs to receive and forward network packets, and also must log packet activity to disk. Then the two tasks are clear: receive/forward, and log. Since logging to disk is a high-latency event, a single-threaded application might incur latency while writing to disk. Splitting into subtasks allows the receive/forward to run without waiting for previous packets to be logged, thus increasing the throughput of the system.

Pattern 3: Multiple Copies of the Same Task. A common variant of multiple independent tasks is multiple copies of the same task (presumably on different data). In this case, we'd require there to be no communication between the different copies, which would enable linear speedup. An example is a rendering application running on multiple distinct animations. We gain throughput, but need to wait just as long for each task to complete.

Pattern 4: Single Task, Multiple Threads. This is the classic vision of "parallelization": for instance, distribute array processing over multiple threads, and let each thread compute the results for a subset of the array.

This pattern, unlike many of the others before it, can actually decrease the time needed to complete a unit of work, since it gets multiple threads involved in doing the single unit simultaneously. The result is improved latency and therefore increased throughput. Communication can be a problem, if the data is not nicely array-structured, or has dependencies between different array parts.

Other names and variations for this pattern include "fork-join", where the main process forks its execution and gives work to all of the threads, with the join synchronizing threads and combining the results; and "divide-and-conquer", where a thread spawns subthreads to compute smaller and smaller parts of the solution.

Pattern 5: Pipeline of Tasks. We've seen pipelining in the context of computer architecture. It can also work for software. For instance, you can use pipelining for packet-handling software, where multiple threads, as above, might confound the order. If you use a three-stage pipeline, then you can have three packets in-flight at the same time, and you can improve throughput by a factor of 3 (given appropriate hardware). Latency would tend to remain the same or be worse (due to communication overhead).

Some notes and variations on the pipeline: 1) if a stage is particularly slow, then it can limit the performance of the entire pipeline, if all of the work has to go through that stage; and 2) you can duplicate pipeline stages, if you know that a particular stage is going to be the bottleneck.

Pattern 6: Client-Server. Botnets work this way (as does `SETI@Home`, etc). To execute some large computation, a server is ready to tell clients what to do. Clients ask the server for some work, and the server gives work to the clients, who report back the results. Note that the server doesn't need to know the identity of the clients for this to work.

A single-machine example is a GUI application where the server part does the backend, while the client part contains the user interface. One could imagine symbolic algebra software being designed that way. Window redraws are an obvious candidate for tasks to run on clients.

Note that the single server can arbitrate access to shared resources. For instance, the clients might all need to perform network access. The server can store all of the requests and send them out in an orderly fashion.

The client-server pattern enables different threads to share work which can somehow be parcelled up, potentially improving throughput. Typically, the parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks. It's also a design pattern that's easy to reason about.

Pattern 7: Producer-Consumer. The producer-consumer is a variant on the pipeline and client-server models. In this case, the producer generates work, and the consumer performs work. An example is a producer which generates rendered frames, and a consumer which orders these frames and writes them to disk. There can be any number of producers and consumers. This approach can improve throughput and also reduces design complexity.

Combining Strategies. If one of the patterns suffices, then you're done. Otherwise, you may need to combine strategies. For instance, you might often start with a pipeline, and then use multiple threads in a particular pipeline stage to handle one piece of data. Or, as I alluded to earlier, you can replicate pipeline stages to handle different data items simultaneously.

Note also that you can get synergies between different patterns. For instance, consider a task which takes 100 seconds. First, you take 80 seconds and parallelize it 4 ways (so, 20 seconds). This reduces the runtime to 40 seconds. Then, you can take the serial 20 seconds and split it into two threads. This further reduces runtime to 30 seconds. You get a $2.5\times$ speedup from the first transformation and $1.3\times$ from the second, if you do it after the first. But, if you only did the second parallelization, you'd only get a $1.1\times$ speedup.

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 [**?**].

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*). 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.

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 and 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. Or 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 functions signature will indicate so. Thus spake Joel[24]:

> *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.* [**?**]

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: $f(x, y, z) \rightarrow (a, b, c)$

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 does encourage the some things that help point you towards functional-style programming. For one thing, it discourages mutability of data, which points you more towards making the arguments to functions be either immutable references (and therefore not changed by the function they are passed to) or ownership transfer (meaning no concurrency). Internal mutability is a thing, but is somewhat discouraged.

[24] "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 [**?**]:

> 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 [**?**].

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 [**?**]. (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 [**?**]) 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 [**?**]: 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 [**?**].

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 [25]. 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 sightly 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

[25] `https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again`

of memory consistency and the different possible reorderings, but for now, just use sequential consistency and you won't get surprises.

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) == false {
        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[26]. 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 [**?**]. 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 [**?**]:

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, though, 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.

[26] `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 [**?**].

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 [**?**]. 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 [**?**], 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 memory accesses follow the sequentially consistent memory model. For instance, if you declared all variables to be C++11 atomics, that would be fine. 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.

Loop- and Memory-Carried Dependencies. We distinguish between *loop-carried* and *memory-carried* dependencies. In a loop-carried dependency, an iteration depends on the result of the previous iteration. For instance, 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.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  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.

Note that you can parallelize the Mandelbrot set calculation by computing the result simultaneously over many points at once. 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.

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:

```
int val = 0;

void g() { val = 1; }
void h() { val = val + 2; }
```

What are the possible outcomes after executing `g()` and `h()` in parallel threads?

A loop-carried dependency is one where an iteration depends on the result of the previous iteration. Let's look at a couple of examples.

Initially, `a[0]` and `a[1]` are 1. Can we run these lines in parallel?

```
a[4] = a[0] + 1;
a[5] = a[1] + 2;
```

`http://www.youtube.com/watch?v=jjXyqcx-mYY`. (This one is legit! Really!)

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 (int i = 1; i < 12; ++i)
    a[i] = a[i-1] + 1;
```

Nope! We can unroll the first two iterations:

```
a[1] = a[0] + 1
a[2] = a[1] + 1
```

Depending on the execution order, either `a[2] = 3` or `a[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*. You would have to play more complicated games to parallelize this.

Now consider this example—is it parallelizable? (Again, all elements initially 1.)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1;
```

Yes, to a degree. We can execute 4 statements in parallel at a time:

- a[4] = a[0] + 1, a[8] = a[4] + 1
- a[5] = a[1] + 1, a[9] = a[5] + 1
- a[6] = a[2] + 1, a[10] = a[6] + 1
- a[7] = a[3] + 1, a[11] = a[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.

Larger loop-carried dependency example. Now consider the following function.

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

How do we parallelize this?

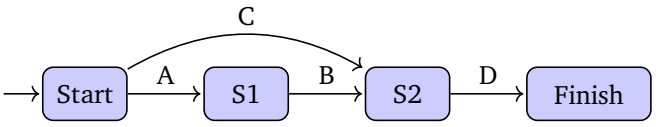
Well, that's a trick question. There's not much that you can do with that function. What you can do is to run this function sequentially for each point, and parallelize along the different points.

As mentioned in class, but one potential problem with that approach is that one point may take disproportionately long. The safe thing to do is to parcel out the work at a finer granularity. There are (unsafe!) techniques for dealing with that too. We'll talk about that later.

Critical Paths

You should be familiar with the concept of a critical path from previous courses; 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.



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 want, then most of the time this is a benefit, and that's what we want.

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:

```
void doWork(int x, int y) {
  int value = longCalculation(x, y);
  if (value > threshold) {
    return value + secondLongCalculation(x, y);
  }
  else {
    return value;
  }
}
```

Without more information, you don't know whether you'll have to execute `secondLongCalculation` or not; it depends on the return value of `longCalculation`.

Fortunately, the arguments to `secondLongCalculation` do not depend on `longCalculation`, so we can call it at any point. Here's one way to speculatively thread the work:

```
void doWork(int x, int y) {
  thread_t t1, t2;
  point p(x,y);
  int v1, v2;
  thread_create(&t1, NULL, &longCalculation, &p);
  thread_create(&t2, NULL, &secondLongCalculation, &p);
  thread_join(t1, &v1);
  thread_join(t2, &v2);
  if (v1 > threshold) {
    return v1 + v2;
  } else {
    return v1;
  }
}
```

We now execute both of the calculations in parallel and return the same result as before.

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 `longCalculation`, the time T_2 that it takes to run `secondLongCalculation`, 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 $T_1 = T_2$.)

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:

```
void doWork(int x, int y) {
  int value = longCalculation(x, y);
  return secondLongCalculation(value);
}
```

If the result of `value` is predictable, then we can speculatively execute `secondLongCalculation` based on the predicted value. (Most values in programs are indeed predictable).

```
void doWork(int x, int y) {
    thread_t t1, t2;
    point p(x,y);
    int v1, v2, last_value;
    thread_create(&t1, NULL, &longCalculation, &p);
    thread_create(&t2, NULL, &secondLongCalculation,
                  &last_value);
    thread_join(t1, &v1);
    thread_join(t2, &v2);
    if (v1 == last_value) {
      return v2;
    } else {
      last_value = v1;
      return secondLongCalculation(v1);
    }
}
```

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 `v1 != 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:

- `longCalculation` and `secondLongCalculation` must not call each other.
- `secondLongCalculation` must not depend on any values set or modified by `longCalculation`.
- The return value of `longCalculation` 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.

Software Transactional Memory

Developers use software transactions by writing `atomic` blocks [**?**]. These blocks are just like `synchronized` blocks, but with different semantics.

```
atomic {
  this.x = this.z + 4;
}
```

You're meant to think of database transactions, which I expect you to 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).

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[27].

```
transfer_funds(Account* sender, Account* receiver, double amount) {
  atomic {
    sender->funds -= amount;
    receiver->funds += amount;
  }
}
```

Using locks, we have two main options:

- Big Global Lock: Lock everything to do with modifying accounts. (This is slow; and you might forget to grab the lock).
- Use a different lock for every account. (Prone to deadlocks; may forget to grab the lock).

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 fa cade doesn't work anymore in the presence of nested transactions.
- 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.

```
atomic {
  x++;
  y++;
}
```

```
atomic {
  if (x != y)
     while (true) { }
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can.

[27] Apparently, bank account transactions aren't actually atomic, but they still make a good example.

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

The formal name for the first idea, quitting early, is is early phase termination [**?**]. 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[28].

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.

[28] 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`. But 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.

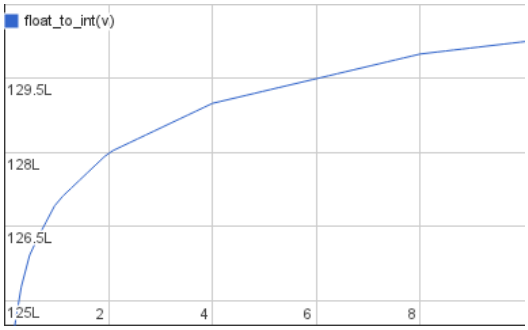
Square root (or similar) is usually calculated by some interpolation or root-finding method (if you took a numerical methods course, you know several techniques for calculating this). But instead there's this [**?**].

```
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, 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 [**?**]:



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, s 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 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 [**?**]. 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 [**?**], 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);
```

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; x86 usually won't (except where there are some specific violations of that; but ARM has weak ordering except where there are data dependencies [**?**]. 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
point.x = 42;
point.y = -42;
point.z = 0;
unlock mutex for point
```

```
lock mutex for point
point.x = 42;
point.y = -42;
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. 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[29]) 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 Data 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:

[29] 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 [**?**].

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 — Rayon

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 naive analysis of the borrowing semantics is that the vector is going to more than one thread. 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. Further research eventually told me to stop reinventing the wheel and use a library for this. Thus, Rayon.

As an aside as to why this – 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. Rayon is, 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.

Back on track. 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.

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[30], 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) [**?**].

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 slides 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[31] 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

[30] Classrooms. How 2019.

[31] You can also compile without SIMD using `-target=i586-unknown-linux-gnu` and see the stack-based x87 instructions.

libraries[32], although we won't do that much. Instead, we'll learn more about how they work and then do some measures 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 [**?**], 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.

Data alignment, however, can be an issue with SIMD. According to [**?**]: "Data must be 16-byte aligned when loading to and storing from the 128-bit XMM registers used by SSE/SSE2/SSE3/SSSE3. This must be done to avoid severe performance penalties.". That's a pretty harsh restriction. SSE4.2 lifts it, if your machine is new enough.

But in any case, Rust will generally align primitives to their sizes. 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/L18`.

```
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.

Case Study on SIMD: Stream VByte

"Can you run faster just by trying harder?"

[32] 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).

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 [**?**], 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 $2^{21} - 1$: $0xxxxxxx/1xxxxxxx/1xxxxxxx$;
- 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 [**?**] 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 does it really mean when you say `-O2`? 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.

Many pages on the Internet describe optimizations. Here's one that contains good examples:

```
http://www.digitalmars.com/ctg/ctgOptimizer.html
```

You can find a full list of `gcc` options here:

```
http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
```

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.

`gcc` **optimization levels.** Here's what `-On` means for `gcc`. Other compilers have similar (but not identical) optimization flags.

- `-O0` (default): Fastest compilation time. Debugging works as expected.
- `-O1` (`-O`): Reduce code size and execution time. No optimizations that increase compiliation time.
- `-O2`: All optimizations except space vs. speed tradeoffs.
- `-O3`: All optimizations.
- `-Ofast`: All `-O3` optimizations, plus non-standards compliant optimizations, particularly `-ffast-math`. (Like `-fast` on the Solaris compiler.)

  This flag turns off exact implementations of IEEE or ISO rules/specifications for math functions. Generally, if you don't care about the exact result, you can use this for a speedup.

Scalar Optimizations

By scalar optimizations, I mean optimizations which affect scalar (non-array) operations. Here are some examples of scalar optimizations.

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 at all optimization levels. 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.

```
a = (c + d) * y;
b = (c + d) * z;

w = 3;
x = f(); y = x;
z = w + y;
```

Enabled at `-O2`*,* `-O3` *or with* `-fgcse`*.* These flags actually enable a global (i.e. across-basic-blocks) CSE pass. This also enables global constant and copy propagation.

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`.

These scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`. More next time.

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:

```
int f(int x) {
  return x * 2;
}
```

```
int 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 particularly profitable when loops execute often. This is often a win, because programs spend a lot of time looping. The trick is to find which loops are going to be the important ones. Profiling is helpful.

A loop induction variable is a variable that varies on each iteration of the loop; the loop variable is definitely a loop induction variable, but there may be others. *Induction variable elimination* gets rid of 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.

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. Here's an example.

```
for (int i = 0; i < 4; ++i)
    f(i)
```
$\Longrightarrow$
```
f(0); f(1); f(2); f(3);
```

Enabled with `-funroll-loops`.

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. For instance, in C:

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        a[j][i] = a[j][i] * c
```
$\Longrightarrow$
```
for (int j = 0; j < M; ++j)
    for (int i = 0; i < N; ++i)
        a[j][i] = a[j][i] * c
```

since C is *row-major* (meaning a[1][1] is beside a[1][2]), rather than *column-major*.

Enabled with `-floop-interchange`.

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...

Loop fusion. This optimization is like the OpenMP collapse construct; we transform

```
for (int i = 0; i < 100; ++i)
    a[i] = 4

for (int i = 0; i < 100; ++i)
    b[i] = 7
```
$\Longrightarrow$
```
for (int i = 0; i < 100; ++i) {
    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 (int i = 0; i < 100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```
$\Longrightarrow$
```
s = x * y;
for (int i = 0; i < 100; ++i) {
    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 two examples.

Branch Prediction. `gcc` attempts to guess the probability of each branch to best order the code. (For an `if`, fall-through is most efficient. Why?)

This isn't quite an optimization, but you can use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program. An example, from the Linux kernel:

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Architecture-Specific. `gcc` can also generate code tuned to particular processors and processor variants. You can specify this using `-march` and `-mtune`. (`-march` implies `-mtune`). This will enable specific instructions that not all CPUs support (e.g. SSE4.2). For example, `-march=corei7`.

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.

Alias and Pointer Analysis

As we've seen in the above analyses, compiler optimizations often need to know about what parts of memory each statement reads to. This is easy when talking about scalar variables which are stored on the stack. This is much harder when talking about pointers or arrays (which can alias). *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.

We've talked about automatic parallelization previously in this course. At this point, I'll remind you that we used `restrict` so that the compiler wouldn't have to do as much pointer analysis. Shape analysis builds on pointer analysis to determine that data structures are indeed trees rather than lists.

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.

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 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:

```
class A {
  public:
    virtual void m();
};

class B : public A {
  public:
    virtual void m() {}
};
```

```
int main(int argc, char *argv[]) {
    std::unique_ptr<A> t(new B);
    t->m();
}
```

Devirtualization could eliminate vtable access; instead, we could just call B's `m` method directly. By the way, "Rapid Type Analysis" analyzes the entire program, observes that only `B` objects are ever instantiated, and enables devirtualization of the `b.m()` call.

Enabled with `-O2`*,* `-O3`*, or with* `-fdevirtualize`.

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.

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything! There are two ways of inlining in C++.

Implicit Inlining (defining a function inside a class definition):

```
class P {
public:
    int get_x() const { return x; }
...
private:
    int x;
};
```

Explicit Inlining:

```
inline max(const int& x, const int& y) {
    return x < y ? y : x;
}
```

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.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors). Just from this your performance may go down easily.

Note also that inlining is merely a suggestion to compilers [**?**]. They may ignore you. For example:

- 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.

Enabled with `-O2` *and* `-O3`.

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 code:

```
int n;

int f() { /* opaque */ }

int main() {
  n = 5;
  f();
  printf("%d\n", n);
}
```

We could propagate the constant value 5, as long as we know that `f()` does not write to `n`.

Tail Recursion Elimination. This optimization is mandatory in some functional languages; we replace a call by a `goto` at the compiler level. Consider this example, courtesy of Wikipedia:

```
int bar(int N) {
  if (A(N))
    return B(N);
  else
    return bar(N);
}
```

For both calls, to `B` and `bar`, we don't need to return control to the calling `bar()` before returning to its caller (because `bar()` is done anyway). This avoids function call overhead and reduces call stack use.

Enabled with `-foptimize-sibling-calls`. Also supports sibling calls as well as tail-recursive calls.

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 [**?**] summarizes recent history. 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 [**?**] 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.

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 [**?**], while gcc details can be found at [**?**].

19 — Optimizing the Compiler

Optimizing the Compiler

20 — Performance Case Studies

Making Firefox Fast

Let's look at Mike Conley's Firefox Performance Updates,

https://mikeconley.ca/blog/2018/02/14/firefox-performance-update-1/

- don't use CPU animating out-of-view elements
- move db init off main thread
- keep better profiling data
- parallel painting for macOS
- lazily instantiate Search Service only when first search starts
- halve size of the blocklist
- refactor to reduce main-thread IO
- don't hold all frames of animated GIFs/APNGs in memory
- eliminate an unnecessary hash table
- use more modern compiler

We can categorize most of these updates into the categories we've seen before:

- do less work
  (or do it sooner/later);
- use threads (move work off main thread);
- track performance;

Which of the updates fall into which categories?

Tab warming

We continue by examining one particular update, *tab warming*, in detail:

https://mikeconley.ca/blog/2018/01/11/making-tab-switching-faster-in-firefox-with-tab-warming/.

"Maybe this is my Canadian-ness showing, but I like to think of it almost like coming in from shoveling snow off of the driveway, and somebody inside has *already made hot chocolate for you*, because they knew youd probably be cold." — Mike Conley

Consider switching tabs. Previously, Firefox would request a paint of the newly-selected tab and wait for the rendering to be available before switching the tab.

The idea is to reduce user-visible latency by predicting an imminent tab switch. How do you know that the user is about to switch tabs? When the user has a mouse, then the mouse cursor will hover over the next tab.

> Assuming a sufficiently long delay between hover and click, the tab switch should be perceived as instantaneous. If the delay was non-zero but still not long enough, we will have nonetheless shaved that time off in eventually presenting the tab to you.
>
> And in the event that we were wrong, and you werent interested in seeing the tab, we eventually throw the uploaded layers away.

The blog post does not report performance numbers (but bug 1430160 discusses how to collect them).

Firefox in general

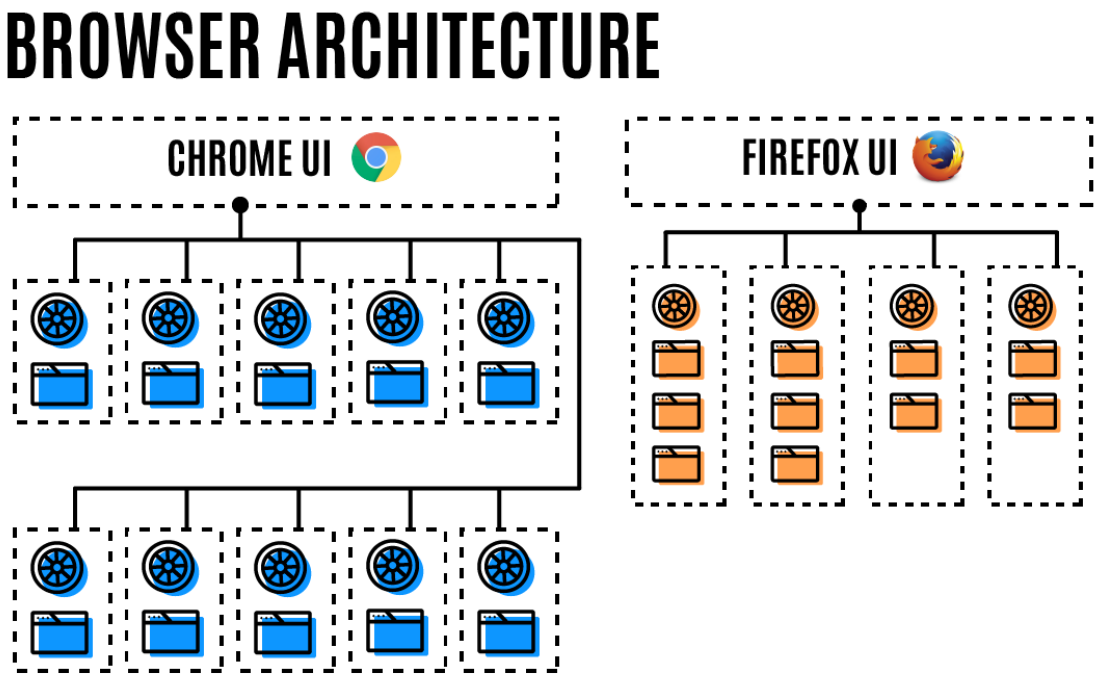
Try: “about:mozilla” in Firefox. On a Quantum Flow-enabled version, you’ll see

The Beast adopted new raiment and studied the ways of Time and Space and Light and the Flow of energy through the Universe. From its studies, the Beast fashioned new structures from oxidised metal and proclaimed their glories. And the Beasts followers rejoiced, finding renewed purpose in these teachings.

from The Book of Mozilla, 11:14

In 2017, Mozilla released Electrolysis (E10s) [**?**]. which leverages multicore processors by using multiple OS-level processes. (Chrome has always done this, but Firefox attempts to also keep memory usage down [**?**]. Beyond internal architecture issues, handling Add-Ons (now WebExtensions) was perhaps the most challenging part of going multi-process.

Note the connection to different thread/process models. Chrome is one-process-per-tab, while Firefox multiplexes tabs across the 4 content processes (“hardware threads”, by analogy). Limiting the number of tabs also limits the memory consumption of the browser: we don’t have arbitrary numbers of renderer state.



Source: Ryan Pollock [**?**]

As a crude summary, Electrolysis works on splitting across processes while the newer Quantum Flow leverages multithreading and other improvements. Quantum Flow uses the Rust programming language and its “fearless

concurrency" (in Rust-speak). Rust should probably be part of a future revision of the ECE 459 curriculum. But we'll focus on Firefox here.

Quantum Flow

Here's a retrospective of the Quantum Flow project:

```
https://ehsanakhgari.org/blog/2017-09-21/quantum-flow-engineering-newsletter-25
```

To sum up, they formed a small team and did the following.

1. Measure slowness: gather information, instrument Firefox, collect profiling data and measurements. Prioritize issues.
2. Gather help: convince other teams to pitch in with perf improvements. Examples: front-end team (reduce flushes, timers); layout team (reflow performance).
3. Fix all the things! (Or at least the most important ones).

Given the short timeline they gave themselves (6 months) and the limited resources, an important part of their work was convincing others to help. They triaged 895 bugs and fixed 369 of them. The weekly Quantum Flow Engineering Newsletter was a key motivational tool.

After the project wound down, they aimed to distribute responsibility for perf improvements across the entire project.

Firefox Telemetry

Firefox's Telemetry feature collects lots of information from Firefox users. Idea: collect data before hacking away at things. Firefox collects hundreds of gigabytes of anonymous metrics per day while browsing and makes it all available to the public. One can view this as an analogy of CPU profiling on a massively distributed context. This data is collected much less often than CPU profiling data but at a much broader scope.

```
https://telemetry.mozilla.org/
```

If you are running Firefox and want to see what it is collecting:

```
about:telemetry
```

You can view distributions of telemetry probes (in the form of histograms). You can also make your own dashboard based on Firefox Telemetry data and Mozilla has infrastructure for their developres to formulate and evaluate their own queries.

Example questions:

- Is Firefox the user's default browser? (69% yes)
- Does e10s make startup faster? (no, slower)
- Which plugins tend to freeze the browser on load? (Silverlight and Flash)

Can see evolution of data over time.

Firefox developers can propose new telemetry probes which are reviewed for data privacy[33] as well as through normal code review channels.

[33] Mozilla Data Collection Practices: `https://wiki.mozilla.org/Firefox/Data_Collection`

Pings. Firefox phones the data home using so-called "pings". Firefox sends a "main ping" every 24 hours, upon shutdown, environment change, and crash. There are other types of pings as well. Pings get sent either by Firefox or by a helper program, Pingsender, when Firefox isn't running. Presumably they are sent over the network as compressed JSON to a central server.

Here's the common ping structure:

```
{
  type: <string>, // "main", "activation", "optout", "saved-session", ...
  id: <UUID>, // a UUID that identifies this ping
  creationDate: <ISO date>, // the date the ping was generated
  version: <number>, // the version of the ping format, currently 4

  application: {
    architecture: <string>, // build architecture, e.g. x86
    buildId: <string>, // "20141126041045"
    name: <string>, // "Firefox"
    version: <string>, // "35.0"
    displayVersion: <string>, // "35.0b3"
    vendor: <string>, // "Mozilla"
    platformVersion: <string>, // "35.0"
    xpcomAbi: <string>, // e.g. "x86-msvc"
    channel: <string>, // "beta"
  },

  clientId: <UUID>, // optional
  environment: { ... }, // optional, not all pings contain the environment
  payload: { ... }, // the actual payload data for this ping type
}
```

Pings contain scalars (counts, booleans, strings) and histograms. A histogram collects bucketed data (think grade distributions). Both scalars and histograms can be keyed, e.g. how often searches happen for which search engines.

Case Study: Is Lower Level Always Faster?

There's a lot of support for the idea that code written in lower level languages (e.g., choosing C rather than C++) means that your code will be faster. Is that always the case? Language elitism aside – not always!

C++11 has made major strides towards readability and efficiency—it provides light-weight abstractions. We'll look at a couple of examples.

Sorting. Our goal is simple: we'd like to sort a bunch of integers. In C, you would usually just use qsort from `stdlib.h`.

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator) (const void*, const void*));
```

This is a fairly ugly definition (as usual, for generic C functions). How ugly is it? Let's look at a usage example.

```
#include <stdlib.h>

int compare(const void* a, const void* b) {
    return (*((int*)a) - *((int*)b));
}

int main(int argc, char* argv[]) {
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

This looks like a nightmare, and is more likely to have bugs than what we'll see next.

C++ has a sort with a much nicer interface[34]:

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp
);
```

It is, in fact, easier to use:

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be a function or a functor. (Don't know what functors are? In C++, they're functions with state.) By default, `sort` uses `operator<` on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

The second question is empirical. Let's see. We generate an array of 2 million ints and sort it (10 times, taking the average).

- qsort: 0.49 seconds
- C++ sort: 0.21 seconds

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

- Custom C sort: 0.29 seconds

Now the C++ version is still faster (but it's close). But, this is quickly going to become a maintainability nightmare.

- Would you rather read a custom sort or 1 line?
- What (who) do you trust more?

[34] . . . well, nicer to use, after you get over templates.

Lesson

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Vectors vs Lists

Consider two problems.

1. Generate **N** random integers and insert them into (sorted) sequence.
   Example: 3 4 2 1
   - 3
   - 3 4
   - 2 3 4
   - 1 2 3 4
2. Remove **N** elements one-at-a-time by going to a random position and removing the element.
   Example: 2 0 1 0
   - 1 2 4
   - 2 4
   - 2
   -

For which **N** is it better to use a list than a vector (or array)?

Complexity analysis. As good computer scientists, let's analyze the complexity.

Vector:

- Inserting
  - $O(\log n)$ for binary search
  - $O(n)$ for insertion (on average, move half the elements)
- Removing
  - $O(1)$ for accessing
  - $O(n)$ for deletion (on average, move half the elements)

List:

- Inserting
  - $O(n)$ for linear search
  - $O(1)$ for insertion
- Removing
  - $O(n)$ for accessing
  - $O(1)$ for deletion

Therefore, based on their complexity, lists should be better.

Reality. OK, here's what happens.

```
$ ./vector_vs_list 50000
Test 1
======
vector: insert 0.1s   remove 0.1s   total 0.2s
list:   insert 19.44s   remove 5.93s   total 25.37s
Test 2
======
vector: insert 0.11s   remove 0.11s   total 0.22s
list:   insert 19.7s   remove 5.93s   total 25.63s
Test 3
======
vector: insert 0.11s   remove 0.1s   total 0.21s
list:   insert 19.59s   remove 5.9s   total 25.49s
```

Vectors dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory. **On 64 bit machines:**
  - Vector: 4 bytes per element.
  - List: At least 20 bytes per element.
- Memory access is slow, and results arrive in blocks:
  - Lists' elements are all over memory, hence many cache misses.
  - A cache miss for a vector will bring a lot more usable data.

So, here are some tips for getting better performance.

- Don't store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.
- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient**.
- Think at a low level if you need to optimize anything.
- Readable code is good code—different hardware needs different optimizations.

21 — Laws of Performance & Performance Culture

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 [**?**].

1. Programming language $<<$ 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.

Performance Tips

While we're on the soapbox, a few more small things to note from [**?**] about how one might write code in a performant as well as maintainable way.

Understand the order of magnitude that matters. If you are writing code where 100 CPU cycles matters, then a function that will acquire a lock through some sort of shared-memory interlocked instruction then this instruction is performance murder. And it might be even worse if lock acquisition fails and you get blocked for 100 000 cycles trying to acquire the lock... On the other hand, if you have a network intensive then the 100 cycle stuff does not matter even the smallest bit because it is lost in the noise of the network.

Plan for the worst case scenario. I don't mean for the nuclear apocalypse, as much fun as that might be. As [**?**] wrote:

> What happens if that lock is held for longer than expected, because the system is under load and the scheduler is overloaded? And what if the owning thread was preempted while holding the lock, and now will not get to run again for quite some time? What happens if the network is saturated because a big news event is underway, or worse, the phone network is intermittently cutting out, the network cable has been unplugged, etc.? What about the case where, because a user has launched far too many applications at once, your memory-intensive operation that usually enjoys nice warmth and locality suddenly begins waiting for the disk on the majority of its memory accesses, due to demand paging? These things happen all the time.
>
> In each of these situations, you can end up paying many more orders of magnitude in cost than you expected under ordinary circumstances. The lock acquisition that usually took 100 CPU cycles now takes several million cycles (as long as a network roundtrip), and the network operation that is usually measured in milliseconds is now measured in tens of seconds, as the software painfully waits for the operation to time out. And your "non-blocking" memory-intensive algorithm on the UI thread just caused a hang, because it's paging like crazy.

No doubt you have seen these sorts of things as well and it is an awful user experience. The spinning beach ball of death and the "Not Responding" added to the title bar are all symptoms of this situation.

Acknowledge and plan for asynchrony. A quick example from [**?**]: suppose that Jordan is writing code to return a list of fonts to be used in the UI. The code checks a local font cache, and if that is already initialized, returns the fonts found in the cache in a List object. If the cache is not initialized, however, the data needs to be loaded, perhaps from the printer (which will dutifully tell you what fonts it can print out). Morgan intends to use this list of fonts in the UI of the program but is unaware that in certain circumstances this call will take much longer (that is, when the cache is not initialized).

Supposing that the API just returns a `List<Font>`, it is not obvious to Morgan where these come from. And in local testing on the dev machine with nothing on it and nothing else doing and the printer is always turned on, and rainbows and unicorns are present. Thus even if the cache is not initialized it is "fast enough". But in the real world, of course, the printer will be down and unicorns are nowhere to be found and it runs up against some hard timeout of 20 seconds.

If, however, Jordan's API now has a return type of `Task<List>` (generify as desired), then it is obvious to Morgan that there is something asynchronous going on here at least some of the time. With this information in hand, the UI won't wait for this task and we might even convince Jordan to give us partial results (so we can draw the UI

elements as backing data arrives) or at the very least, a progress bar. Users love progress bars. They are wildly inaccurate and not super informative even if they are accurate, but at least they give the user the impression that something is happening.

That actually leads us down a side debate about how to make the program appear faster than it is by some UI trickery and psychological "hacks", but that is beyond what we want to talk about here...

Performance Culture

Now let's take it up a level: how do you make performance an important aspect of the culture of development of a project. The source for this section is [**?**].

The author's recommendations of warning signs that performance culture is off the rails:

- Answering the question, "how is the product doing on my key performance metrics," is difficult.
- Performance often regresses and team members either don't know, don't care, or find out too late to act.
- Blame is one of the most common responses to performance problems (either people, infrastructure, or both).
- Performance tests swing wildly, cannot be trusted, and are generally ignored by most of the team.
- Performance is something one, or a few, individuals are meant to keep an eye on, instead of the whole team.
- Performance issues in production are common, and require ugly scrambles to address (and/or cannot be reproduced).

As tempting as it is to say these are all technical issues, they are really human problems. It would obviously be preferable to start with a good performance culture in the first place, but that is not always going to happen. But suppose you are already in a hole. The usual course of action is to stop and say to yourself "I appear to be in a hole... I know, I'll dig my way out!". Maybe I am cynical. The right thing to do would be stop digging, of course...

Change has to come from both the top down and the bottom up. Management needs to make performance a priority: ask questions, demand rigour... while at the same time, people doing the development need to understand the performance of what they write, making practice improvements. Both sides need to have zero tolerance for regression.

If only one person is responsible for performance, this will not work. One person could not reasonably keep up with the rest of the team. If the other developers don't have performance in mind when writing, it would mean the key performance person would spend unnecessary time rewriting things when the it could have been right in the first place. If this isn't convincing, think about unit tests. Could you really "outsource" the writing of all unit tests to one person? And would code quality be the same if you did?

The author is very adamant that poor performance culture is management's fault. Managers need to budget time, reward the work, and encourage performance work. Managers who don't understand performance culture are likely to be caught by surprise and blame things.

Consider the following comparison from [**?**]:

> Manager A gives lip service to performance culture. She, however, packs every sprint schedule with a steady stream of features – "we've got to crush competitor Z and must reach feature parity!" – with no time for breaks in-between. She spends all-hands team meetings praising new features, demos aplenty, and even gives out a reward to an engineer at each one for "the most groundbreaking feature." As a result, her team cranks out features at an impressive clip, delivers fresh demos to the board every single time, and gives the sales team plenty of ammo to pursue new leads. There aren't performance gates and engineers generally don't bother to think much about it.
>
> Manager B takes a more balanced approach. She believes that given the competitive landscape, and the need to impress customers and board members with whizbang demos, new features need to keep

> coming. But she is also wary of building up too much debt in areas like performance, reliability, and quality for areas she expects to stick. So she intentionally puts her foot on the brake and pushes the team just as hard on these areas as she does features. She demands good engineering systems and live flighting of new features with performance telemetry built-in, for example. This requires that she hold board members and product managers at bay, which is definitely unpopular and difficult. In addition to a reward for "the most groundbreaking feature" award at each all-hands, she shows charts of performance progress and delivers a "performance ninja" award too, to the engineer who delivered the most impactful performance improvement. Note that engineering systems improvements also qualify!

But this is a startup, you object, why does it matter, because we need the minimum viable product and we needed it yesterday, and also, VCs like features. Well maybe, but architecture matters. There's time to fix things later, maybe, but you can really cripple your startup by choosing the wrong things early on.

After a discussion of the why and the what, it is time to think about how to make it systematic. If it is not systematic, it will not get done and it will be dropped when in a time crunch. And development teams are always in a time crunch! This is why best practices have automated unit tests that run on every commit (or at least build). And much the same way, there need to be performance tests; if something has significantly regressed performance, the build failed and the commit is no good. This obviously requires that the tests be meaningful, testing effectively and not overly noisy.

Performance tests may be a bit too large and difficult to run for every commit or every build. Although it would be nice to know exactly which commit is the cause of the problem, you may want a test that runs overnight to see the long term trends. And thus a 12-hour test is probably unsuitable for this. The solution is, obviously, levels of test.

One company I worked for did something like this, although it was not performance specific. Before committing to the cvs repository (at this point nobody had heard of git and it was the newest thing) there was a set of tests called "precheckin". Nobody forced a developer to run it, but you were definitely supposed to, and it just verified that everything would compile and some basic stuff worked. These tests took a few minutes to run. Then each build on the build server had a larger set of tests ("build"), including the precheckin material and a lot more, taking about 30-40 minutes to execute. Breaking this breaks the build (and gets everyone's attention). Then there were the "regression" tests that ran overnight (and ALL night).

In [**?**] a zero tolerance rules advocated: anything that significantly regresses performance is reverted and re-worked. No exceptions, no questions, no pleading one's case, etc. If exceptions are allowed, exceptions quickly become the rule, and promises to fix something later are procrastinated endlessly.

Obviously when code does more it should be expected to take longer, but then the test(s) should be adjusted as well. This is the same as when developing new features/changes, after all; unit tests are adjusted when code changes are introduced intentionally.

Metrics There are, roughly speaking, two categories of metrics:

- *Consumption*: measure resources consumed by running a test.
- *Observational*: measure the outcomes of running the test, from the view of outside the system.

Consumption metrics are the things that we can easily measure about the code run during the test: cache misses, TLB misses, context switches, number of I/Os, memory allocated, number of system calls (traps are interrupts, remember). Observational metrics are more along the lines of time it takes to run a dataset, or number of work items processed per unit time, etc.

It might seem like observational metrics are all that we care about. End users care about the amount of time it takes to complete their work, or management/sales/marketing/C-levels care about the throughput and that's it. This matters, sure, but it's not enough to know the outcome; the consumption metrics are necessary to break down why the total time (or throughput) is what it is.

Suppose a change is committed and the tests run and the total time it takes increases by 10%. If that's all you know, you know that things got worse but have no data as to why. You can obviously read the source code or the

diff and try to look for it, but even that might not tell you enough. If you see that memory allocations are reported and you have now allocated more memory than before, at least you have a theory to start with on what's slowing down the code. It might not be that, it might be something else entirely, but it's a place to start. Memory allocation changes are relatively easy to spot though: there's a new call to `malloc` or a structure has been extended in some way. If it is a cache miss issue, finding that without the data is harder...

Does this remind you of `printf` debugging? It's a little bit like that. But then again, you want to use this sort of thing for long running tests anyway. If the test is going to run for 4h it would be helpful to know where exactly in that four hours it went off the rails (on a crazy train!).

There's one more consideration and that is the variability between runs. In the ideal situation, there is consistent performance every run. That is probably too much to ask for and there will be some natural variation due to nondeterminism in computers. But a change that causes a wild change in the variance is also no good; a test that sometimes finishes in $0.5x$ time and sometimes in $2x$ time has a high variance and this will not be acceptable.

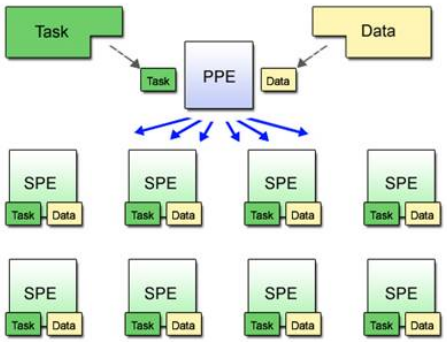
Be ready to dig deep: suppose a developer has introduced a change that is slow under rare circumstances. Those rare circumstances are not triggered until n years later when a second developer introduces another change. The second change will be identified, at least initially, as the "problem". There's two ways to go about it; either fix the initial change or work around it. Which one is correct will depend very much on the circumstances, but at the very least both options should be considered. It's not about blame or not about fault; it's about improving the performance of the software.

22 — 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 OpenCL (i.e. Open Computing Language). 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 [**?**] 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:

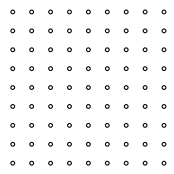


(from the Linux Cell documentation)

CUDA (Compute Unified Device Architecture) is NVIDIA's architecture for processing on GPUs. "C for CUDA" predates OpenCL; NVIDIA still 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 do this in OpenCL, but it seems to be the case that CUDA has got widespread acceptance out in industry, so we have chosen to 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.

Programming Model. The programming model for all of these architectures is similar: write the code for the massively parallel computation (kernel) separately from the main code, transfer the data to the GPU coprocessor (or execute it on the CPU), wait, then transfer the results back.

OpenCL includes both task parallelism and data parallelism, as we've discussed earlier in this course. *Data parallelism* is central to OpenCL; in OpenCL's view, you are evaluating a function, or *kernel*, at a set of points, like so:



Another name for the set of points is the *index space*. Each of the points corresponds to a *work-item*.

OpenCL also supports *task parallelism*: it can run different kernels in parallel. Such kernels may have a one-point index space. The documentation doesn't say much about task parallelism.

More on work-items. The work-item is the fundamental unit of work in OpenCL. 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. OpenCL spawns a thread for each work item, with a unique thread ID. The system runs each work-group on a set of cores; NVIDIA calls that set a *warp*, while ATI calls it a *wavefront*. The scheduler assigns work-items to the warps/wavefronts until there are no more work items left.

Shared memory. OpenCL 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; like a user-managed cache;
- global memory: shared between all work-items as well as the host;
- constant memory: resides on the GPU, and cached. Does not change.

There is also host memory, which generally contains the application's data.

An example kernel. Let's continue by looking at a sample kernel, first written traditionally and then written as an OpenCL kernel [**?**].

```
void traditional_mul(int n, const float *a, const float *b, float *c) {
  int i;
  for (i = 0; i < n; i++) c[i] = a[i] * b[i];
}
```

The same code looks like this as a kernel:

```
kernel void opencl_mul(global const float *a, global const float *b, global float *c) {
  int id = get_global_id(0);  // dimension 0
  c[id] = a[id] * b[id];
}
```

You can write kernels in a variant of C. OpenCL takes away some features, like function pointers, recursion, variable-length arrays, bit fields, and standard headers; and adds work-items, workgroups, vectors, synchronization, and declarations of memory type. OpenCL also provides a library for kernels to use.

Branches. OpenCL implements a SIMT architecture. What this means is that the computation for each work-item can branch arbitrarily. The hardware will execute all branches that any thread in a warp executed (which can be slow).

```
kernel void contains_branch(global float *a, global
     float *b) {
    int id = get_global_id(0);
    if (cond) {
        a[id] += 5.0;
    } else {
        b[id] += 5.0;
    }
}
```

In the above example, the `if` statement will cause each thread to execute both branches of the `if`, keeping only the result of the appropriate branch.

Similarly, executing a loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item.

```
kernel void contains_loop(global float *a,
                          global float *b) {
    int id = get_global_id(0);

    for (i = 0; i < id; i++) {
        b[i] += a[i];
    }
}
```

If you're setting up workgroups, though, you can arrange for all of the work-items in a workgroup to execute the same branches.

Synchronization. You might define workgroups because you can only put barriers and memory fences between work items in the same workgroup. Different workgroups execute independently.

OpenCL also supports all of the notions that we've talked about before: memory fences (read and write), barriers, and the volatile keyword. The barrier (`barrier()`) ensures that all of the threads in the workgroup all reach the barrier before they continue. Recall that the fence ensures that no load or store instructions (depending on the type of fence) migrate to the other side of the fence.

Complete OpenCL Example

```
// Note by PL: don't use this example as a template; it uses the C bindings!
// Instead, use the C++ bindings as in the other example.
// source: pages 1-9 through 1-11, http://developer.amd.com/wordpress/media/2013/07/
     AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

//
// Copyright (c) 2010 Advanced Micro Devices, Inc. All rights reserved.
//
// A minimalist OpenCL program.
#include <CL/cl.h>
#include <stdio.h>
#define NWITEMS 512

// A simple memset kernel
const char *source =
"__kernel void memset( __global uint *dst )                             \n"
"{                                                                      \n"
"   dst[get_global_id(0)] = get_global_id(0);                           \n"
"}                                                                      \n";

int main(int argc, char ** argv)
{
   // 1. Get a platform.
   cl_platform_id platform;
   clGetPlatformIDs( 1, &platform, NULL );

   // 2. Find a gpu device.
   cl_device_id device;
   clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU,
```

```
                1,
                &device,
                NULL);

    // 3. Create a context and command queue on that device.
    cl_context context = clCreateContext( NULL,
                                          1,
                                          &device,
                                          NULL, NULL, NULL);
    cl_command_queue queue = clCreateCommandQueue( context,
                                                   device,
                                                   0, NULL );
    // 4. Perform runtime source compilation, and obtain kernel entry point.
    cl_program program = clCreateProgramWithSource( context,
                                                    1,
                                                    &source,
                                                    NULL, NULL );
    clBuildProgram( program, 1, &device, NULL, NULL, NULL );
    cl_kernel kernel = clCreateKernel( program, "memset", NULL );
    // 5. Create a data buffer.
    cl_mem buffer = clCreateBuffer( context,
                                    CL_MEM_WRITE_ONLY,
                                    NWITEMS * sizeof(cl_uint),
                                    NULL, NULL );
    // 6. Launch the kernel. Let OpenCL pick the local work size.
    size_t global_work_size = NWITEMS;
    clSetKernelArg(kernel, 0, sizeof(buffer), (void*) &buffer);
    clEnqueueNDRangeKernel( queue,
                            kernel,
                            1,       // dimensions
                            NULL,    // initial offsets
                            &global_work_size, // number of work-items
                            NULL,    // work-items per work-group
                            0, NULL, NULL);  // events
    clFinish( queue );
    // 7. Look at the results via synchronous buffer map.
    cl_uint *ptr;
    ptr = (cl_uint *) clEnqueueMapBuffer( queue,
                                          buffer,
                                          CL_TRUE,
                                          CL_MAP_READ,
                                          0,
                                          NWITEMS * sizeof(cl_uint),
                                          0, NULL, NULL, NULL );
    int i;
    for(i=0; i < NWITEMS; i++)
        printf("%d %d\n", i, ptr[i]);
    return 0;
}
```

Walk-through. Let's look at all of the code in the example and explain the terms. 1) First, we request an OpenCL *platform*. Platforms, also known as hosts, contain 2) OpenCL *compute devices*, which may in turn contain multiple compute units. Note that we could also request a CPU device in step 2, without changing the rest of the code.

Next, in step 3, we request an OpenCL *context* (representing all OpenCL state) and create a *command-queue*. We will request that OpenCL do work by telling it to run a kernel in the queue.

In step 4, we create an OpenCL *program*. This is a confusing term; an OpenCL program is what runs on the compute unit, and includes kernels, functions, and declarations. Your application can contain more than one OpenCL program. In this case, we create a program from the C string `source`, which contains the kernel `memset`. OpenCL can also create programs from binaries, which may be in an intermediate representation, or already compiled for a particular device. We get a pointer to the kernel in this step, as the return value from `clCreateKernel`.

There's one more step before launching the kernel; in step 5, we create a *data buffer*, which enables communication between devices. Recall that OpenCL requires explicit communication, which we'll see later. Since this example doesn't have input, we don't need to put anything into the buffer initially.

Finally, we can launch the kernel in step 6. In this case, we don't specify anything about workgroups, but enqueue the entire 1-dimensional index space, starting at (0). We also state that the index space has `NWITEMS` elements, and

not to subdivide the problem into work-items. The last three parameters are about events. We call `clFinish()` to wait for the command-queue to empty.

Finally, in step 7, we copy the results back from the shared buffer using `clEnqueueMapBuffer`. This copy is blocking (first `CL_TRUE` argument), so we don't need an explicit `clFinish()` call. We also indicate the details of the command we'd like to run: in particular, a read of `NWITEMS` from the buffer.

You might also want to consider cleaning up the objects you've allocated; I haven't shown that here. The code also doesn't contain any error-handling.

It's important to note that this is a large amount of setup and accordingly overhead cost. Doing the setup, copying the data in, waiting, and copying the data out, all take time. 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.

C++ Bindings. If we use the C++ bindings, we'll get automatic resource release and exceptions. C++ likes to use the RAII style (resource allocation is initialization).

- Change the header to `CL/cl.hpp` and define `__CL_ENABLE_EXCEPTIONS`.

We'd also like to store our kernel in a file instead of a string. The C API is not so nice to work with; the C++ API is nicer. Use it! As an example, you'll find the `vector_add` example in the slides and in the code repo.

More Complicated Kernel

I've omitted the C code. it's pretty similar to what we saw before, but it uses workgroups, customized to the number of compute units on the device. Here is a more interesting kernel, also from the same source.

```
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable

// 9. The source buffer is accessed as 4-vectors.
__kernel void minp( __global uint4 *src,
                    __global uint  *gmin,
                    __local uint   *lmin,
                    __global uint  *dbg,
                    size_t          nitems,
                    uint            dev )
{
  // 10. Set up __global memory access pattern.
  uint count  = ( nitems / 4 ) / get_global_size(0);
  uint idx    = (dev == 0) ? get_global_id(0) * count
                           : get_global_id(0);
  uint stride = (dev == 0) ? 1 : get_global_size(0);
  uint pmin   = (uint) -1;

  // 11. First, compute private min, for this work-item.
  for( int n=0; n < count; n++, idx += stride )
  {
    pmin = min( pmin, src[idx].x );
    pmin = min( pmin, src[idx].y );
    pmin = min( pmin, src[idx].z );
    pmin = min( pmin, src[idx].w );
  }
```

```
  // 12. Reduce min values inside work-group.
  if( get_local_id(0) == 0 )
    lmin[0] = (uint) -1;
  barrier( CLK_LOCAL_MEM_FENCE );

  (void) atom_min( lmin, pmin );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Write out to __global.
  if( get_local_id(0) == 0 )
    gmin[ get_group_id(0) ] = lmin[0];

  // Dump some debug information.
  if( get_global_id(0) == 0 )
     { dbg[0] = get_num_groups(0); dbg[1] = get_global_size(0);
       dbg[2] = count; dbg[3] = stride; }
}

// 13. Reduce work-group min values from __global to __global.
__kernel void reduce( __global uint4 *src, __global uint *gmin )
{
   (void) atom_min( gmin, gmin[get_global_id(0)] ) ;
}
```

Let's discuss the notable features of this code, which finds the minimum value from an array of 32-bit ints. (OpenCL ints are always 32 bits). Steps 1 through 8 are in the C code, which I've omitted; see the AMD guide for the code. At 9), we can investigate the signature of the `minp` kernel. The use of `uint4`, or 4-int vectors, enables SSE instructions on CPUs and helps out GPUs as well. We'll access the constituent `int`s of `src` using the `.x`, `.y`, `.z` and `.w` fields. This kernel also writes to an array of global minima, `gmin`, and an array of local minima (inside the workgroup), `lmin`.

In step 10, we figure out where our point in the index space, as reported by `get_global_id()`, is located in the `src` index, as well as the stride, which is 1 for CPUs and $7 \times 64 \times c$, where c is the number of work units, which was rounded up using the following heuristic:

```
  cl_uint ws = 64;
  global_work_size = compute_units * 7 * ws; // 7 wavefronts per SIMD
  while ( (num_src_items / 4) % global_work_size != 0 )
    global_work_size += ws;

  local_work_size = ws;
```

The core of the kernel occurs in step 11, where the `for`-loop computes the local minimum of the array elements in the work-item. In this stage, we are reading from the `__global` array `src`, and writing to the private memory `pmin`. This takes almost all of the bandwidth.

Then, in stage 12, thread 0 of the workgroup initializes the workgroup-local `lmin` value, and each thread atomically compares (using the extended atomic requested using the pragma) its `pmin` to the local `lmin` value. We have local memory fences here to make sure that threads stay in synch. This code is not going to consume much memory bandwidth, since there aren't many threads per work-group, and there's only local communication.

Finally, thread 0 of the workgroup writes the local minimum of the workgroup to the global array `gmin`. In step 13, a second kernel traverses the `gmin` array and finds the smallest minimum.

Summary. We've now seen the basics of GPU programming. The key idea is to define a kernel and find a suitable index space. Then you execute the kernel over the index space and collect results. The main difficulty is in formulating your problem in such a way that you can parallelize it, and then in splitting it into workgroups.

C++ Bindings Example

Use the C++ bindings. They're better.

```
// Vector add example, C++ bindings (use these!)
// source:
//  http://www.thebigblob.com/getting-started-
```

```
//        with-opencl-and-gpu-computing/

#define __CL_ENABLE_EXCEPTIONS

#include <CL/cl.hpp>

#include <iostream>
#include <fstream>
#include <string>
#include <utility>
#include <vector>

int main() {
    // Create the two input vectors
    const int LIST_SIZE = 1000;
    int *A = new int[LIST_SIZE];
    int *B = new int[LIST_SIZE];
    for(int i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }

    try {
        // Get available platforms
        std::vector<cl::Platform> platforms;
        cl::Platform::get(&platforms);

        // Select the default platform and create a context
        // using this platform and the GPU
        cl_context_properties cps[3] = {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[0])(),
            0
        };
        cl::Context context(CL_DEVICE_TYPE_GPU, cps);

        // Get a list of devices on this platform
        std::vector<cl::Device> devices =
            context.getInfo<CL_CONTEXT_DEVICES>();

        // Create a command queue and use the first device
        cl::CommandQueue queue = cl::CommandQueue(context,
            devices[0]);

        // Read source file
        std::ifstream sourceFile("vector_add_kernel.cl");
        std::string sourceCode(
            std::istreambuf_iterator<char>(sourceFile),
            (std::istreambuf_iterator<char>())
        );
        cl::Program::Sources source(
            1,
            std::make_pair(sourceCode.c_str(),
                           sourceCode.length()+1)
        );

        // Make program of the source code in the context
        cl::Program program = cl::Program(context, source);

        // Build program for these specific devices
        program.build(devices);

        // Make kernel
        cl::Kernel kernel(program, "vector_add");

        // Create memory buffers
        cl::Buffer bufferA = cl::Buffer(
            context,
            CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(int)
        );
        cl::Buffer bufferB = cl::Buffer(
            context,
            CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(int)
        );
```

```
        cl::Buffer bufferC = cl::Buffer(
            context,
            CL_MEM_WRITE_ONLY,
            LIST_SIZE * sizeof(int)
        );

        // Copy lists A and B to the memory buffers
        queue.enqueueWriteBuffer(
            bufferA,
            CL_TRUE,
            0,
            LIST_SIZE * sizeof(int),
            A
        );
        queue.enqueueWriteBuffer(
            bufferB,
            CL_TRUE,
            0,
            LIST_SIZE * sizeof(int),
            B
        );

        // Set arguments to kernel
        kernel.setArg(0, bufferA);
        kernel.setArg(1, bufferB);
        kernel.setArg(2, bufferC);

        // Run the kernel on specific ND range
        cl::NDRange global(LIST_SIZE);
        cl::NDRange local(1);
        queue.enqueueNDRangeKernel(
            kernel,
            cl::NullRange,
            global,
            local
        );

        // Read buffer C into a local list
        int* C = new int[LIST_SIZE];
        queue.enqueueReadBuffer(
             bufferC,
             CL_TRUE,
             0,
             LIST_SIZE * sizeof(int),
             C
        );

        for(int i = 0; i < LIST_SIZE; i ++) {
            std::cout << A[i] << " + " << B[i] << " = "
                      << C[i] << std::endl;
        }
    } catch(cl::Error error) {
        std::cout << error.what() << "(" << error.err()
                  << ")" << std::endl;
    }

    return 0;
}
```

23 — Password Cracking, Bitcoin Mining

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 [**?**]. See also `http://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 user 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.

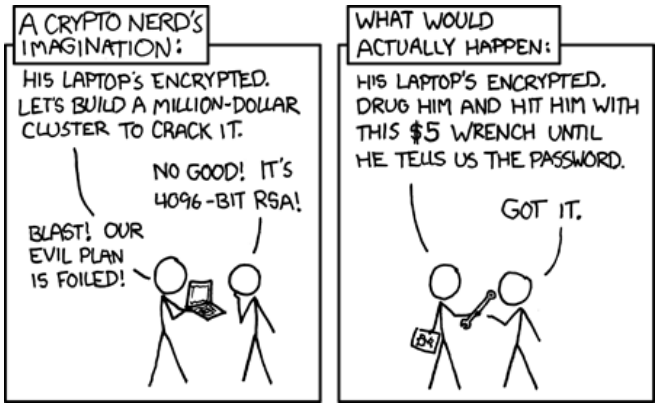
Arms race: making cracking difficult. The idea has always been to make it more difficult to compute the hash function. This does make it longer for the user when they want to log in, but the amount of time to compute a single password is reasonable. However, it's intractable to try all possible passwords, 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.

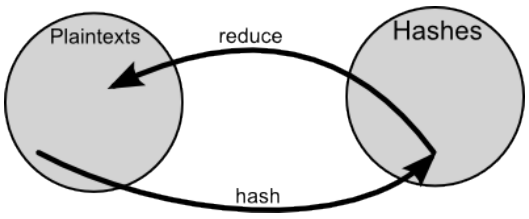
Existence proof. Of course anyone can define anything. It's much better if the thing being defined actually exists. The scrypt paper then goes on to exhibit ReMix, which is a concrete example of a sequential memory hard function.

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, 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*.

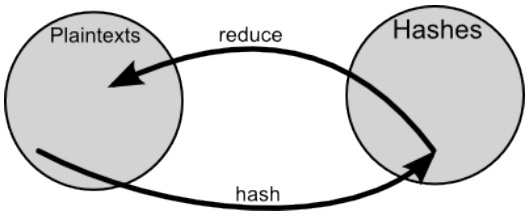
There is a technical paper describing how rainbow tables work, but we'll instead use a much less cryptograhic-expert-level explanation [**?**].

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 [**?**].

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 [**?**].

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 [**?**]:

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 Denmark and it uses electricity comparable to the entire power consumption of the country of Austria (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 [**?**], 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.

24 — Profiling

Profiling

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[35] but we should really follow one of my favourite rules: "don't guess, measure!"[36] 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 your assignment 2...?).

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...

There is always the "informal" way of doing this; it sort of works but it's not exactly the best plan. You probably know that when developing a program you can "debug" it without using any tools (e.g., `gdb`) by inserting a lot of print statements to the console or the log file. So when you enter function `foo` you print a nice little line on the console that say something like "entering function foo", associated with a timestamp and then when you're ready to return, a corresponding print function that says "exiting" appears, also with a timestamp.

This approach kind of works, and I've used it myself to figure out what blocks of a single large function are taking a long time (updating exchange rates... yeah). 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 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.

[35] There is a saying that even a blind squirrel sometimes finds a nut.
[36] Now I am certain you are sick of hearing that.

(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.)

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), query the system state; or,
- instrumentation-based, or probe-based/predicate-based (traditionally too expensive): query system state under certain conditions; like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling.

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"[37].

Tips for Leveraging Profiling. When writing large software projects:

- First, write clear and concise code.
  Don't do any premature optimizations—focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you 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;
- 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 [**?**].

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.

[37] `http://en.wikipedia.org/wiki/Unobtainium`

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 `gprof`, the GNU profiler tool[38], and then continue with other tools.

`gprof` does sampling-based profiling for single processes: it requests that the operating system interrupt the process being profiled at regular time intervals and figures out which procedure is currently running. It also adds a bit of instrumentation to collect information about which procedures call other procedures.

"Flat" profile. The obvious thing to do with the profile information is to just print it out. You get a list of procedures called and the amount of time spent in each of these procedures.

The general limitation is that procedures that don't run for long enough won't show up in the profile. (There's a caveat: if the function was compiled for profiling, then it will show up anyway, but you won't find out about how long it executed for).

"Call graph". `gprof` can also print out its version of a call graph, which shows the amount of time that either a function runs (as in the "flat" profile) as well as the amount of time that the callees of the function run. Another term for such a call graph is a "dynamic call graph", since it tracks the dynamic behaviour of the program. Using the `gprof` call graph, you can find out who is responsible for calling the functions that take a long time.

Limitations of `gprof`. Beyond the usual limitations of a process-oriented profiler, `gprof` also suffers limitations from running completely in user-space. That is, it has no access to information about system calls, including time spent doing I/O. It also doesn't know anything about the CPU's built-in counters (e.g. cache miss counts, etc). Like the other profilers, it causes overhead when it's running, but the overhead isn't too large.

`gprof` usage guide

We'll give some details about using `gprof`. First, use the `-pg` flag with `clang` when compiling and linking. (It's currently broken in gcc, unless you give `-no-pie`.) Next, run your program as you normally would. Your program will now create `gmon.out`.

Use gprof to interpret the results: `gprof <executable>`.

Example. Consider a program with 100 million calls to two math functions.

[38] `http://sourceware.org/binutils/docs/gprof/`

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}

int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Looking at the code, we have no idea what takes longer. One might guess that floating point math takes longer. This is admittedly a silly example, but it works well to illustrate our point.

Flat Profile Example. When we run the program and look at the flat profile, we see:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

There is one function per line. Here are what the columns mean:

- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.
- **calls:** number of times this function was called.
- **self ns/call:** just self nanoseconds / calls.
- **total ns/call:** mean function execution time, including calls the function makes.

Call Graph Example. After the flat profile gives you a feel for which functions are costly, you can get a better story from the call graph.

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000     int_math [2]
                0.43    6.04 100000000/100000000     float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000     main [1]
```

```
[2]     53.2    0.58    7.13 100000000         int_math [2]
                2.44    3.13 100000000/100000000     int_math_helper [4]
                1.56    0.00 100000000/300000000     int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000     main [1]
[3]     44.7    0.43    6.04 100000000         float_math [3]
                1.65    2.93 100000000/100000000     float_math_helper [6]
                1.47    0.00 100000000/300000000     float_power [7]
-----------------------------------------------
                2.44    3.13 100000000/100000000     int_math [2]
[4]     38.4    2.44    3.13 100000000         int_math_helper [4]
                3.13    0.00 200000000/300000000     int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000     int_math [2]
                3.13    0.00 200000000/300000000     int_math_helper [4]
[5]     32.4    4.69    0.00 300000000         int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000     float_math [3]
[6]     31.6    1.65    2.93 100000000         float_math_helper [6]
                2.93    0.00 200000000/300000000     float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000     float_math [3]
                2.93    0.00 200000000/300000000     float_math_helper [6]
[7]     30.3    4.40    0.00 300000000         float_power [7]
```

To interpret the call graph, note that the line with the index [N] is the *primary line*, or the current function being considered.

- Lines above the primary line are the functions which called this function.
- Lines below the primary line are the functions which were called by this function (children).

For the primary line, the columns mean:

- **time:** total percentage of time spent in this function and its children.
- **self:** same as in flat profile.
- **children:** time spent in all calls made by the function;
  - should be equal to self + children of all functions below.

For callers (functions above the primary line):

- **self:** time spent in primary function, when called from current function.
- **children:** time spent in primary function's children, when called from current function.
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function.

For callees (functions below the primary line):

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - self + children is an estimate of time spent in current function when called from primary function.
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function.

Based on this information, we can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc. This example isn't too exciting; we could simplify the math and optimize the program that way.

Introduction to gperftools

Next, we'll talk about the Google Performance Tools.

```
http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html
```

They include:

- a CPU profiler
- a heap profiler
- a heap checker; and
- a faster (multithreaded) `malloc`.

We'll mostly use the CPU profiler. Characteristics include:

- supposedly works for multithreaded programs;
- purely statistical sampling;
- no recompilation required (typically benefit from re-linking); and
- better output, including built-in graphical output.

You can use the profiler without any recompilation. But this is not recommended; you'll get worse data. Use `LD_PRELOAD`, which changes the dynamic libraries that an executable uses.

```
% LD_PRELOAD="/usr/lib/libprofiler.so" CPUPROFILE=test.prof ./test
```

The other (more-recommended) option is to link to the profiler with `-lprofiler`.

Both options read the `CPUPROFILE` environment variable, which specifies where profiling data goes.

You can use the profiling library directly as well:

```
#include <gperftools/profiler.h>
```

Then, bracket code you want profiled with:

```
ProfilerStart()
// ...
ProfilerStop()
```

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable (default value 100 interrupts/second).

`pprof` **usage.** `pprof` is like `gprof` for Google Perf Tools. It analyzes profiling results. Here are some usage examples.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output `dot`, `ps`, `pdf` or `gif` instead of `gv`.

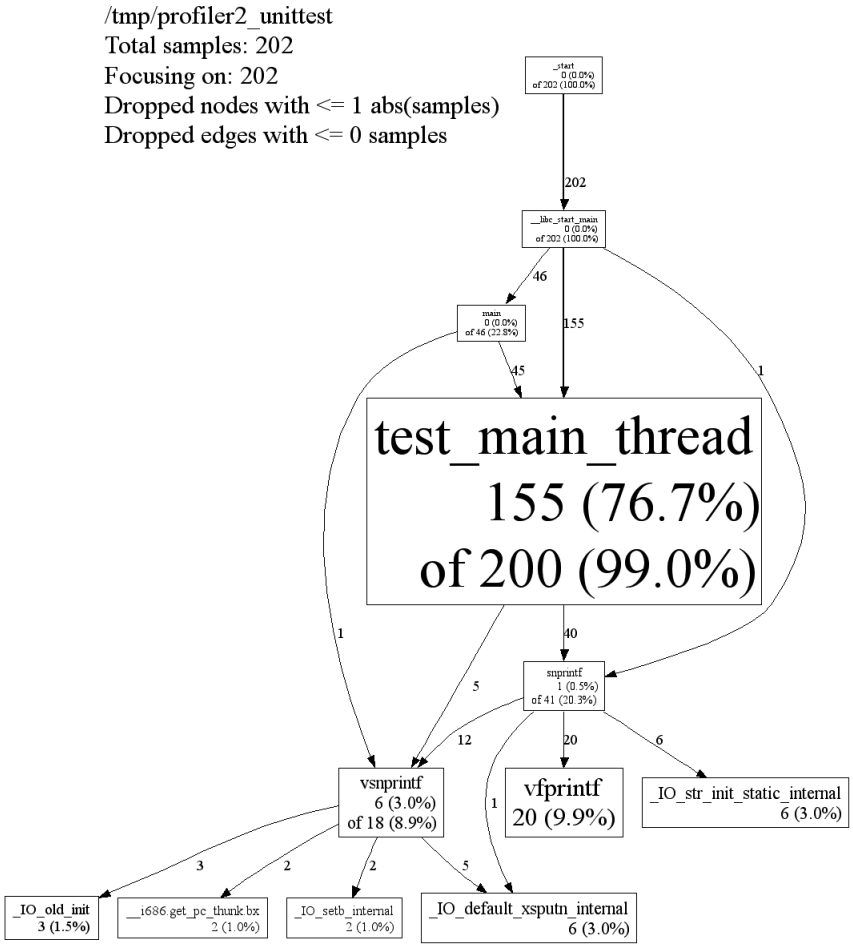
`pprof` **text output.** This is similar to the flat profile in `gprof`.

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Columns, from left to right, denote:

- Number of samples in this function.
- Percentage of samples in this function (same as **time** in `gprof`).
- Percentage of checks in the functions printed so far (equivalent to **cumulative**, but in %).
- Number of checks in this function and its callees.
- Percentage of checks in this function and its callees.
- Function name.

Graphical Output. Google Perf Tools can also produce graphical output:



This shows the same numbers as the text output. This particular graphical example is on a different program than the text output, and that's why they look different. Directed edges denote function calls. Note:

$$\#\text{ of samples in callees} = \#\text{ in ``this function + callees''} - \#\text{ in ``this function''}.$$

For example, in `float_math_helper`, we have "51 (local) of 96 (cumulative)". Here,

$$96 - 51 = 45 \text{ (callees)}.$$

- callee `int_power` = 7 (bogus)
- callee `float_power` = 38
- callees total = 45

Note that the call graph is not exact. In fact, it shows many bogus relations which clearly don't exist. For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `-focus` flag to look at individual functions.

25 — System-Level Profiling, Profiler Guided Optimization

System-level profiling

Most profiling tools interrogate the CPU in more detail than `gprof` and friends. These tools are typically aware of the whole system, but may focus on one application, and may have both per-process and system-wide modes. We'll discuss a couple of these tools here, highlighting conceptual differences between these applications.

Solaris Studio Performance Analyzer. This tool[39] supports `gprof`-style profiling ("clock-based profiling") as well as kernel-level profiling through DTrace (described later). At process level, it collects more process-level data than `gprof`, including page fault times and wait times. It also can read CPU performance counters (e.g. the number of executed floating point adds and multiplies). As a Sun application, it also works with Java programs.

Since locks and concurrency are important, modern tools, including the Studio Performance Analyzer, can track the amount of time spent waiting for locks, as well as statistics about MPI message passing. More on lock waits below, when we talk about WAIT.

VTune. Intel and AMD both provide profiling tools; Intel's VTune tool costs money, while AMD's CodeAnalyst tool is free software.

Intel uses the term "event-based sampling" to refer to sampling which fires after a certain number of CPU events occur, and "time-based sampling" to refer to the `gprof`-style sampling (e.g. every 100ms). VTune can also correlate the behaviour of the counters with other system events (like disk workload). Both of these sampling modes also include the behaviour of the operating system and I/O in their counts.

VTune also supports an instrumentation-based profiling approach, which measures time spent in each procedure (same type of data as `gprof`, but using a different collection scheme).

VTune will also tell you what it thinks the top problems with your software are. However, if you want to understand what it's saying, you do actually need to understand the architecture.

CodeAnalyst. AMD also provides a profiling tool. Unlike Intel's tool, AMD's tool is free software (the Linux version is released under the GPL), so that, for instance, Mozilla suggests that people include CodeAnalyst profiling data when reporting Firefox performance problems [40].

CodeAnalyst is a system-wide profiler. It supports drilling down into particular programs and libraries; the only disadvantage of being system-wide is that the process you're interested in has to execute often enough to show up in the profile. It also uses debug symbols to provide meaningful names; these symbols are potentially supplied over the Internet.

Like all profilers, it includes a sampling mode, which it calls "Time-based profiling" (TBP). This mode works on

[39] You can find a high-level description at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-performance-tools-183986.pdf`

[40] `https://developer.mozilla.org/Profiling_with_AMD_CodeAnalyst`

all processors. The other modes are "Event-based profiling" (EBP) and "Instruction-based sampling" (IBS); these modes use hardware performance counters.

AMD's CodeAnalyst documentation points out that your sampling interval needs to be sufficiently high to capture useful data, and that you need to take samples for enough time. The default sampling rate is once every millisecond, and they suggest that programs should run for at least 15 seconds to get meaningful data.

The EBP mode works like VTune's event-based sampling: after a certain number of CPU events occur, the profiler records the system state. That way, it knows where e.g. all the cache misses are occuring. A caveat, though, is that EBP can't exactly identify the guilty statement, because of "skid": in the presence of out-of-order execution, guilt gets spread to the adjacent instructions.

To improve the accuracy of the profile information, CodeAnalyst uses AMD hardware features to watch specific x86 instructions and "ops", their associated backend instructions. This is the IBS mode[41] of CodeAnalyst. AMD provides an example[42] where IBS tracks down the exact instruction responsible for data translation lookaside buffer (DTLB) misses, while EBP indicates four potential guilty instructions.

oprofile. This free software is a sampling-based tool which uses the Linux Kernel Performance Events API to access CPU performance counters. It tracks the currently-running function (or even the line of code) and can, in system-wide mode, work across processes, recording data for every active application.

Webpage: `http://oprofile.sourceforge.net`.

You can run oprofile either in system-wide mode (as root) or per-process. To run it in system-wide mode:

```
% sudo opcontrol --vmlinux=/usr/src/linux-3.2.7-1-ARCH/vmlinux
% echo 0 | sudo tee /proc/sys/kernel/nmi_watchdog
% sudo opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

Or, per-process:

```
[plam@lynch nm-morph]$ operf ./test_harness
operf: Profiler started

Profiling done.
```

Both of these invocations produce profiling output. You can read the profiling output by running `opreport` and giving it your executable.

```
% sudo opreport -l ./test
CPU: Intel Core/i7, speed 1595.78 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not
halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        symbol name
7550     26.0749  int_math_helper
5982     20.6596  int_power
5859     20.2348  float_power
3605     12.4504  float_math
3198     11.0447  int_math
2601      8.9829  float_math_helper
160       0.5526  main
```

If you have debug symbols (`-g`) you can also get better data:

```
% sudo opannotate --source --output-dir=/path/to/annotated-source /path/to/mybinary
```

Use `opreport` by itself for a whole-system view. You can also reset and stop the profiling.

[41] Available on AMD processors as of the K10 family—typically manufactured in 2007+; see `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`. Thanks to Jonathan Thomas for pointing this out.

[42] `http://developer.amd.com/cpu/CodeAnalyst/assets/ISPASS2010_IBS_CA_abstract.pdf`

```
% sudo opcontrol --reset
Signalling daemon... done
% sudo opcontrol --stop
Stopping profiling.
```

perf. This uses the same base data as oprofile, but provides a better (git-like) interface. Once again, it 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 last year:

```
[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
```

perf can tell you which instructions are taking time, or which lines of code; compile with `-ggdb` to enable source code viewing.

```
% perf record ./test_harness
% perf annotate
```

`perf annotate` is interactive. Play around with it.

DTrace. DTrace [**?**] is an instrumentation-based system-wide profiling tool designed to be used on production systems. It supports custom queries about system behaviour: when you are debugging system performance, you can collect all sorts of data about what the system is doing. The two primary design goals were in support of use in production: 1) avoid overhead when not tracing and 2) guarantee safety (i.e. DTrace can never cause crashes).

DTrace runs on Solaris and some BSDs. There is a Linux port, which may be usable. I'll try to install it on `ece459-1`.

Probe effect. "Wait! Don't 'instrumentation-based' and 'production systems' not go together?" For instance, Valgrind incurs a 100× slowdown. Ouch.

Nope! DTrace was designed to have zero overhead when inactive. It does this by dynamically rewriting the code to insert instrumentation when requested. So, if you want to instrument all calls to the `open` system call, then DTrace is going to replace the instruction at the beginning of `open` with an unconditional branch to the instrumentation code, execute the profiling code, then return to your code. Otherwise, the code runs exactly as if you weren't looking.

Safety. As I've mentioned before, crashing a production system is a big no-no. DTrace is therefore designed to never cause a system crash. How? The instrumentation you write for DTrace must conform to fairly strict constraints.

DTrace system design. The DTrace framework supports instrumentation *providers*, which make *probes* (i.e. instrumentation points) available; and *consumers*, which enable probes as appropriate. Examples of probes include system calls, arbitrary kernel functions, and locking actions. Typically, probes apply at function entry or exit points. DTrace also supports typical sampling-based profiling in the form of timer-based probes; that is, it executes instrumentation every 100ms. This is tantamount to sampling.

You can specify a DTrace clause using probes, predicates, and a set of action statements. The action statements execute when the condition specified by the probe holds and the predicate evaluates to true. D programs consist of a sequence of clauses.

Example. Here's an example of a DTrace query from [**?**].

```
syscall::read:entry {
        self->t = timestamp;
}

syscall::read:return
/self->t/ {
        printf("%d/%d spent %d nsecs in read\n"
            pid, tid, timestamp - self->t);
}
```

The first clause instruments all entries to the system call `read` and sets a thread-local variable `t` to the current time. The second clause instruments returns from `read` where the thread-local variable `t` is non-zero, calling `printf` to print out the relevant data.

The D (DTrace clause language) design ensures that clauses cannot loop indefinitely (since they can't loop at all), nor can they execute unsafe code; providers are responsible for providing safety guarantees. Probes might be unsafe because they might interrupt the system at a critical time. Or, action statements could perform illegal writes. DTrace won't execute unsafe code.

Workflow. Both the USENIX article [**?**] and the ACM Queue article [**?**] referenced above contain example usages of DTrace. In high-level terms: first identify a problem; then, use standard system monitoring tools, plus custom DTrace queries, to collect data about the problem (and resolve it).

WAIT

Another approach which recently appeared in the research literature is the WAIT tool out of IBM. Unfortunately, this tool is not free and not generally available. Let's talk about it anyways.

Like DTrace, WAIT is suitable for use in production environments. It uses hooks built into modern Java Virtual Machines (JVMs) to analyze their idle time. It performs a sampling-based analysis of the behaviour of the Java VM. Note that its samples are quite infrequent; they suggest that taking samples once or twice a minute is enough. At each sample, WAIT records the state of each of the threads, which includes its call stack and participation in system locks. This data enables WAIT to compute (using expert rules) an abstract "wait state". The wait state indicates what the process is currently doing or waiting on, e.g. "disk", "GC", "network", or "blocked".

Workflow. You run your application, collect data (using a script or manually), and upload the data to the server. The server provides a report which you use to fix the performance problems. The report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

The paper presents six case studies where WAIT helped solve performance problems, including deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Other Applications of Profiling.

Profiling applies to languages beyond C/C++ and Java, of course. If you are profiling an interpreted language, you'll need a specific tool to get useful results. For Python, you can use `cProfile`; it is a standard implementation of profiling, from what I can see.

Here's a short tangent. Many of the concepts that we've seen for code also apply to web pages. Google's Page Speed tool[43], in conjunction with Firebug, helps profile web pages, and provides suggestions on how to make your web pages faster. Note that Page Speed includes improvements for the web page's design, e.g. not requiring multiple DNS lookups; leveraging browser caching; or combining images; as well as traditional profiling for the JavaScript on your pages.

Profiler Guided Optimization (POGO)

In 2015 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). Try as I might, I was not able to convince him to fly in just for this lecture.

The compiler does a great deal of 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 from [?]:

```
void whichBranchIsTaken(int a, int b) {
       if (a < b) {
              printf(" a is less than b. \n");
       } else {
              printf(" b is greater than or equal to a. \n");
       }
}

void devirtualization(int count) {
   for (int i = 0; i < count; i++) {
          (*p) (x, y);
   }
}


void switchCaseExpansion(int i) {
       switch(i) {
              case 1:
                     printf(" Case 1 was chosen \n");
                     break;
              case 2:
                     printf(" Case 2 was chosen \n");
                     break;
       }
}
```

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.

There are similar questions raised for the other two examples. What is the "normal" value for some pointer `p`? If we do not know, the compiler cannot do devirtualization (replace this virtual call with a real one). Same thing with `i`: what is its typical value? If we know that, it is our prediction. Actually, in a switch-case 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 C(++) (Sparta) and we don't have that: 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,

[43] `http://code.google.com/speed/page-speed/`

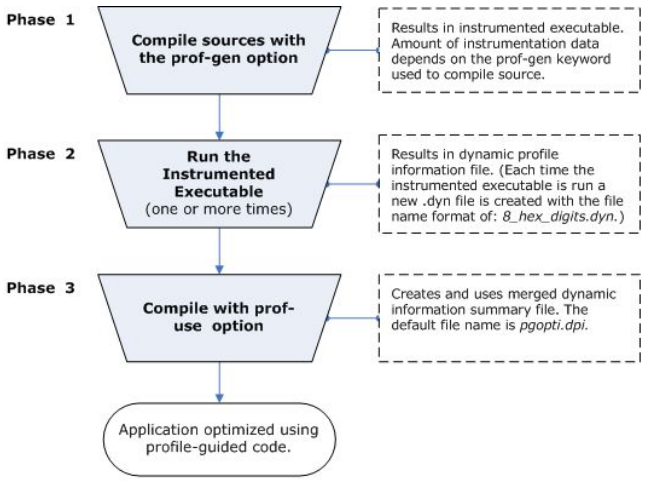
then recompile the code using the data we collected. Let's expand on all three steps.

Step one is to generate an executable with instrumentation. 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 `switch` statement. When this phase is complete, there is an instrumented executable and an empty database file where the training data goes [**?**].

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. 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...

Step three is a recompile. This time, in addition to the source files, the training data is fed into the compiler for a second compile, and this data is applied to produce a better output executable than could be achieved by static analysis alone.

The Intel Developer Zone explains the process in a handy infographic[44] :



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 [**?**], 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.

What does it mean for it 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 [**?**] 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. Obviously 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 multiple runs can be merged with user assigned weightings.

[44] Source: `https://software.intel.com/en-us/node/522721`

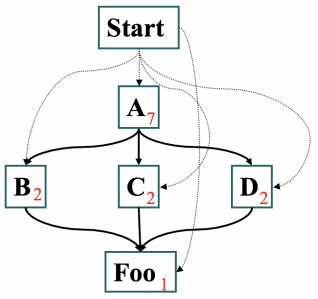
Behind the Scenes

In the optimize phase, the training data is used to do the following optimizations [?]:

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

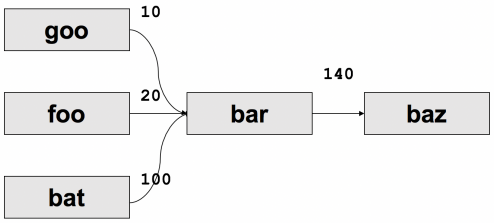
For the most part we should be familiar with the techniques that are listed as being other compiler optimizations we have previously discussed. The new ones are (3) speed and size decision, which we have just covered; and items (4) and (5) which relate to how to pack the generated code in the binary.

According to [?] 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`[45]. Great, let's look at this call graph from [?]:

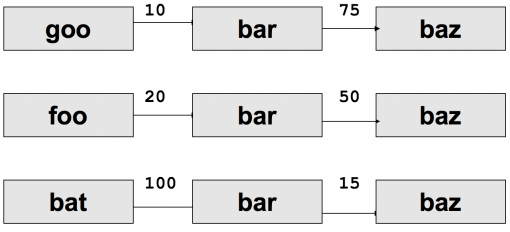


Quick analysis of this code would have us find all the ways in which the functions might call each other. In total, there are 14 paths in this code, seven of which get us to function `Foo`.

Consider another diagram showing the relationships between functions, in which the numbers on the edges represent the number of invocations [?]:

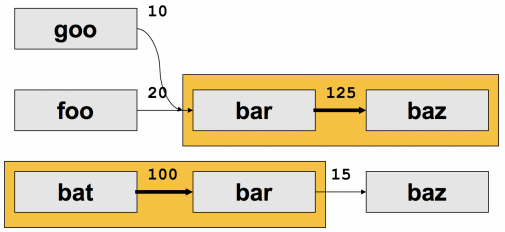


When considering what to do here, POGO takes the view like this [?]:

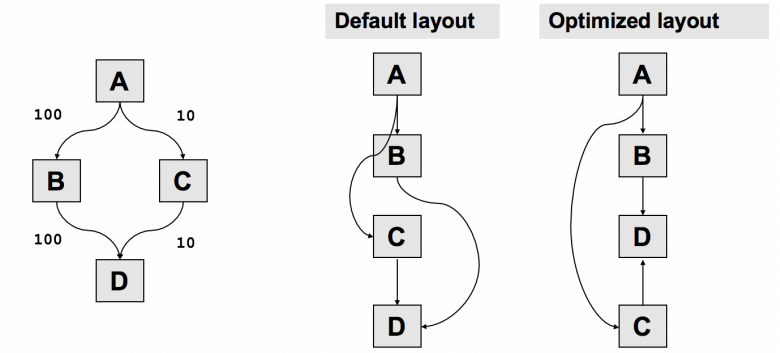


[45] Why are these the example function names?! It's kind of like Alice and Bob...

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 [**?**]:



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 [**?**]:



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 [**?**] summarizes the gains to be made. The application under test is a standard benchmark suite (Spec2K):

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.

26 — Liar, Liar

Lies about Calling Context

Let's open with a video that illustrates one of the problems with sampling-based profiling:

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, however, is that the camera is taking images at some multiple of the frequency of the blade rotation speed so it gives the illusion that the blades are not spinning at all. This is a sampling problem, and you see the same problem in car commercials on TV where it looks like the wheels are spinning backwards. They're not, but the sampling effect of the camera can make it look that way.

Some profiler results are real. Other results are interpolated, and perhaps wrong. Who can we trust? We'll start by talking about gprof and callgrind/KCacheGrind. The reference for this part of the lecture is a blog post by Yossi Kreinin [**?**].

Running Example. Consider the following code.

```
void work(int n) {
  volatile int i=0; //don't optimize away
  while(i++ < n);
}
void easy() { work(1000); }
void hard() { work(1000*1000*1000); }
int main() { easy(); hard(); }
```

We see that there is a worker function whose runtime depends on its input. Function `easy` calls the worker function with a small input, and `hard` calls it with a large input. So we expect most of the time should be spent in the `hard` function, right? Profiling yields:

```
[plam@lynch L27]\$ gprof ./try gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  \%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
101.30      1.68     1.68        2   840.78   840.78  work
  0.00      1.68     0.00        1     0.00  {840.78}  easy
  0.00      1.68     0.00        1     0.00  {840.78}  hard
```

Most of the profiler output is just fine. But there are lies in the "total ms/call" column. The call to `easy` takes about 0 seconds, while that to `hard` takes 1.68s. Less importantly, the total ms/call for `work` is indeed an average, but that hides the variance between runtimes.

Why? To make any sense of the lies, we need to understand how `gprof` works. It uses two standard-library functions: **profil()** and **mcount()**.

- **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. Bringing that information back to the profiler output columns, we can see that the "calls" column is reliable; the "self seconds" column is sampled, but reasonably accurate here; and the "total ms/call" is interpolated, and we deceived it in this contrived example. How is that?

gprof sees:

- a total of 1.68s in `work`;
- 1 call to `work` from `easy`; and
- 1 call to `work` from `hard`.

All of these numbers are reliable. However, gprof draws the unreliable inference that both `easy`, `hard` cause 840ms of `work` time. How did we get here? Well, gprof used some correct data: 3.84 seconds spent in `work`, divides it between `easy` and `hard`, each of which `mcount` reported was called once, and gets 1.92. So we got a bogus result using correct

This is wrong. `work` takes 1000000× longer when called from `hard`!

The following results from gprof are suspect:

- contribution of children to parents;
- total runtime spent in self+children;
- etc.

When are call graph edges right? Two cases:

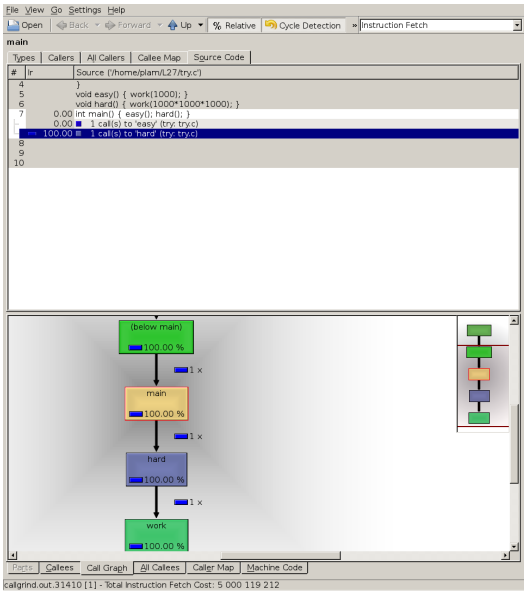
- functions with only one caller (e.g. `f()` only called by `g()`); or,
- functions which always take the same time to complete (e.g. `rand()`).

On the other hand, results for any function whose running time depends on its inputs, and which is called from multiple contexts, are sketchy.

callgrind/KCacheGrind

Next, we'll talk about callgrind/KCacheGrind. Like our old friends memcheck, hell grind, and cachegrind, callgrind is part of valgrind, and runs the program under an x86 JIT. KCacheGrind is a frontend to callgrind. callgrind gives better information, but imposes more overhead.

KCacheGrind works properly on the earlier running example:



It properly reports that `hard` takes all the time. But we can still deceive it.

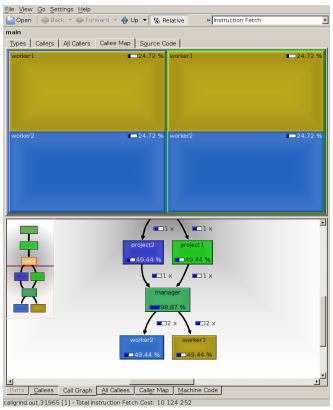
More Complex Example. Let's look at this example.

```
void worker1(int n) {
  volatile int i=0;
  while(i++<n);
}
void worker2(int n) {
  volatile int i=0;
  while(i++<n);
}
void manager(int n1, int n2) {
  worker1(n1);
  worker2(n2);
}
```

```
void project1() {
  manager(1000, 1000000);
}
void project2() {
  manager(1000000, 1000);
}
int main() {
  project1();
  project2();
}
```

Now, `worker2` takes all the time in `project1`, and `worker1` takes all the time in `project2`.

Let's see how KCacheGrind does on this example.



The call graph, on the bottom, shows that `worker1` and `worker2` do each take about 50% of time. So do `project2` and `project1`. This is fine. (I think gprof would correctly interpolate that too.)

However, KCacheGrind also lies, in the report on top. It is saying that `worker1` and `worker2` doing half the work in each project. That's not what the code says. Why is it lying?

- gprof reports time spent in `f()` and `g()`, and how many times `f()` calls `g()`.
- callgrind also reports time spent in `g()` when called from `f()`, i.e. some calling-context information.
- callgrind does *not* report time spent in `g()` when called from `f()` when called from `h()`. We don't get the `project1` to `manager` to `worker1` link. (We have Edges but need Edge-Pairs).

Summary. We've seen that some profiler results are exact; some results are sampled; and some results are interpolated. If you understand the tool, you understand where it can go wrong.

Understand your tools!

Lies from Metrics

While app-specific metrics can lie too, mostly we'll talk about CPU perf counters today.

The reference here is a blog post by Paul Khuong [**?**].

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 = better than expected! Hmm.

The next step is (as we do in this course) 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.

```
# 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>
```

We can see that 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)
    [...]
    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.

Metrics lie, though, and when you focus on the metrics as opposed to what you actually care about, it's easy to be led astray.

In this case, what we actually care about is the total # of cycles.

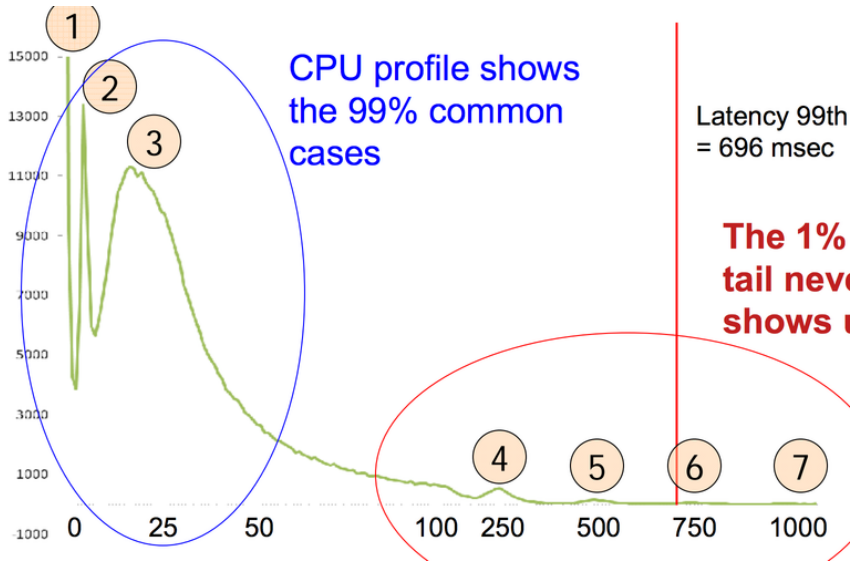
No atomic/fence:	2.81e9 cycles
lock inc/dec:	3.66e9 cycles
mfence:	19.60e9 cycles

That 15% number is a total lie. Profilers, even using CPU expense 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.

The Long Tail

Our source here is the blog post by Dan Luu [**?**]. 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:

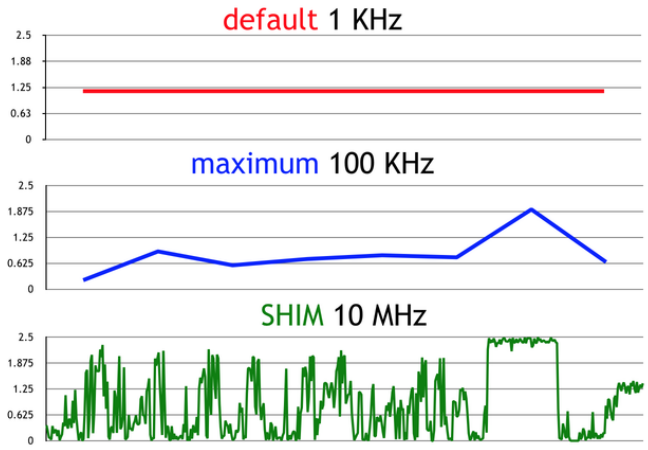


Let's break it down. Peak 1 corresponds to something cached in RAM—best case scenario. Peak 2 is at around 3ms, which is too fast for spinning and seeking magnetic hard disks, but it's fast enough for reading something from the disk cache via the PCI-Express interface. Peak 3 is obviously disk seek and read times, somewhere 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 [**?**]: 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 [**?**], 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 doing useful work. SHIM gets around this by being more invasive—it adds 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.

Algorithmic profiling. Coppa et al [**?**] have proposed another profiling tool, `aprof`. `https://code.google.com/p/aprof/`

> aprof is a Valgrind tool for performance profiling designed to help developers discover hidden asymptotic inefficiencies in the code. From one or more runs of a program, aprof measures how the performance of individual routines scales as a function of the input size, yielding clues to its growth rate and to the "big-O" of the program.

Summary We saw a bunch of lies today: 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.

27 — Memory Profiling, Cachegrind

~~Memory Profiling~~ Return to Asgard

Thus far we have focused on CPU profiling. Other kinds of profiling got some mention, but they're not the only kind of profiling we can do. Memory profiling is also a thing, and specifically we're going to focus on heap profiling. We kind of touched on the subject a little bit earlier when we looked at finding memory leaks. The ideas are the same: we don't want to leak memory, but remember that last category (other than suppressed), "Still Reachable", things that remained allocated and we still had pointers to them, but were not properly deallocated? Right, we care about them too, and for that we want to do heap profiling.

If we don't look after those things, we're just using more and more memory over time. That likely means more paging and the potential for running out of heap space altogether. Again, the memory isn't really lost, because we could free it.

Well, let's start with where we left off. Returning to the realm of Asgard, we're going to call again on our old friend Valgrind. Except this time we're going to use a fourth tool in it: Massif. This is, obviously, a joke on "massive", combined with the name Sif, a Norse goddess associated with the earth (and in the Marvel movies, Shieldmaiden to Thor). While we're on the subject, Sif has an axe (shield?) to grind with Loki, because at some point he cut off her golden hair (and in the Marvel films, it grew back in dark). That Loki—what a trickster! Right, we're digressing... what do you mean the course isn't ECE 459: Norse Mythology?!

So what does Massif do? It will tell you about how much heap memory your program is using, and also how the situation got to be that way. So let's start with the example program from the documentation [**?**]:

```
#include <stdlib.h>

void g ( void ) {
    malloc( 4000 );
}

void f ( void ) {
    malloc( 2000 );
    g();
}

int main ( void ) {
    int i;
    int* a[10];

    for ( i = 0; i < 10; i++ ) {
        a[i] = malloc( 1000 );
    }
    f();
    g();

    for ( i = 0; i < 10; i++ ) {
        free( a[i] );
    }
    return 0;
}
```

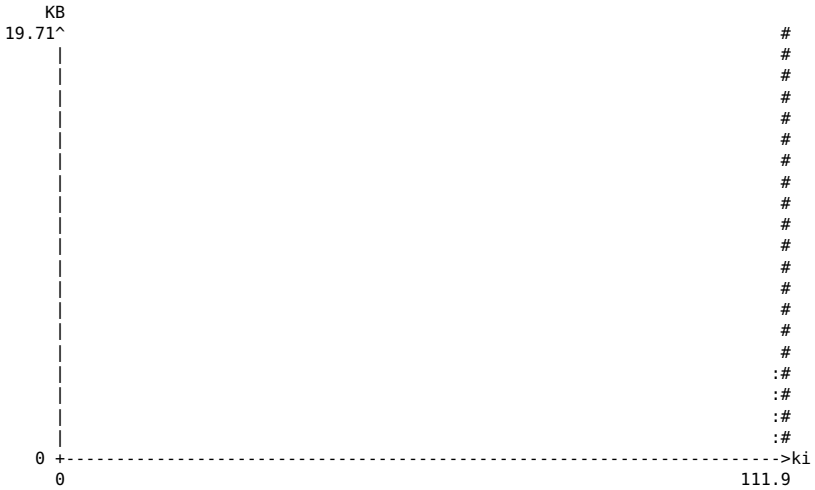
After we compile (remember the `-g` option for debug symbols), run the command:

```
jz@Loki:~/ece459$ valgrind --tool=massif ./massif
```

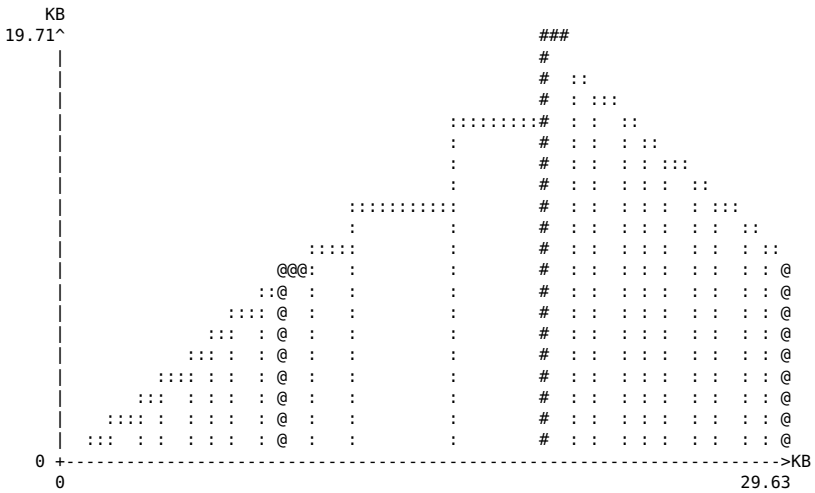
```
==25187== Massif, a heap profiler
==25187== Copyright (C) 2003-2013, and GNU GPL'd, by Nicholas Nethercote
==25187== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==25187== Command: ./massif
==25187==
==25187==
```

Doesn't that look useful?! What happened? Your program executed slowly, as is always the case with any of the Valgrind toolset, but you don't get summary data on the console like we did with Valgrind or helgrind or cachegrind. Weird. What we got instead was the file `massif.out.25187` (matches the PID of whatever we ran). This file, which you can open up in your favourite text editor is not especially human readable, but it's not incomprehensible like the output from cachegrind ("Aha, a 1 in column 4 of line 2857. That's what's killing our performance!"). There is an associated tool for summarizing and interpreting this data in a much nicer way: `ms_print`, which has nothing whatsoever to do with Microsoft. Promise.

If we look at the output there (hint: pipe the output to `less` or something, otherwise you get a huge amount of data thrown at the console), it looks much more user friendly.



Now wait a minute. This bar graph might be user friendly but it's not exactly what I'd call... useful, is it? For a long time, nothing happens, then... kaboom! According to the docs, what actually happened here is, we gave in a trivial program where most of the CPU time was spent doing the setup and loading and everything, and the trivial program ran for only a short period of time, right at the end. So for a relatively short program we should tell Massif to care more about the bytes than the CPU cycles, with the `--time-unit=B` option. Let's try that.



Neat. Now we're getting somewhere. We can see that 25 snapshots were taken. It will take snapshots whenever there are appropriate allocation and deallocation statements, up to a configurable maximum, and for a long running program, toss some old data if necessary. Let's look in the documentation to see what the symbols mean (they're not just to look pretty). So, from the docs [**?**]:

- Most snapshots are normal (they have just basic information) They use the ':' characters.
- Detailed snapshots are shown with '@' characters. By default, every 10th snapshot is detailed.
- There is at most one peak snapshot. The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. The peak snapshot is represented in the graph by a bar consisting of '#' characters.

As a caveat, the peak can be a bit inaccurate. Peaks are only recorded when a deallocation happens. This just avoids wasting time recording a peak and then overwriting it; if you are allocating a bunch of blocks in succession (e.g., in assignment 1, a bunch of structs that have a buffer) then you would constantly be overwriting the peak over and over again. Also, there's some loss of accuracy to speed things up. Well, okay.

So let's look at the snapshots. We'll start with the normal ones. There are 9 of those, numbers 0 through 8:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0             0            0
  1          1,016            1,016          1,000            16            0
  2          2,032            2,032          2,000            32            0
  3          3,048            3,048          3,000            48            0
  4          4,064            4,064          4,000            64            0
  5          5,080            5,080          5,000            80            0
  6          6,096            6,096          6,000            96            0
  7          7,112            7,112          7,000           112            0
  8          8,128            8,128          8,000           128            0
```

The columns are pretty much self explanatory, with a couple exceptions. The time(B) column corresponds to time measured in allocations thanks to our choice of the time unit at the command line. The extra-heap(B) represents internal fragmentation[46] in the blocks we received. The stacks column shows as zero because by default, Massif doesn't look at the stack. It's a heap profiler, remember?

Number 9 is a "detailed" snapshot, so I've separated it out, and reproduced the headers there to make this a little easier to remember what they are.

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  9          9,144            9,144          9,000           144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x4005BB: main (massif.c:17)
```

So the additional information we got here is a reflection of where our heap allocations took place. Thus far, all the allocations took place on line 17 of the program, which was `a[i] = malloc( 1000 );` inside that for loop.

Then let's look at the peak snapshot (again, trimmed a bit to call out exactly what we need to see here):

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
```

[46] Remember from operating systems: if the user asked for some n bytes where n is not a nice multiple the returned block may be "rounded up". So a request for 1000 bytes is bumped up to 1016 bytes in this example. The extra space is "wasted" but it's nicer than having a whole bunch of little tiny useless fragments of the heap to be managed.

```
 14         20,184           20,184           20,000           184            0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x4005BB: main (massif.c:17)
|
->39.64% (8,000B) 0x400589: g (massif.c:4)
| ->19.82% (4,000B) 0x40059E: f (massif.c:9)
| | ->19.82% (4,000B) 0x4005D7: main (massif.c:20)
| |
| ->19.82% (4,000B) 0x4005DC: main (massif.c:22)
|
->09.91% (2,000B) 0x400599: f (massif.c:8)
  ->09.91% (2,000B) 0x4005D7: main (massif.c:20)
```

Massif has found all the allocations in this program and distilled them down to a tree structure that traces the path through which all of these various memory allocations occurred. So not just where the `malloc` call happened, but also how we got there.

When program termination occurs we get a final output of what blocks remains allocated and where they come from. These point to memory leaks, incidentally, and valgrind would not be amused with us.

```
 24         30,344           10,024           10,000            24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x400589: g (massif.c:4)
| ->39.90% (4,000B) 0x40059E: f (massif.c:9)
| | ->39.90% (4,000B) 0x4005D7: main (massif.c:20)
| |
| ->39.90% (4,000B) 0x4005DC: main (massif.c:22)
|
->19.95% (2,000B) 0x400599: f (massif.c:8)
| ->19.95% (2,000B) 0x4005D7: main (massif.c:20)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

In fact, if I ask valgrind what it thinks of this program, it says:

```
jz@Loki:~/ece459$ valgrind ./massif
==25775== Memcheck, a memory error detector
==25775== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==25775== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==25775== Command: ./massif
==25775==
==25775==
==25775== HEAP SUMMARY:
==25775==     in use at exit: 10,000 bytes in 3 blocks
==25775==   total heap usage: 13 allocs, 10 frees, 20,000 bytes allocated
==25775==
==25775== LEAK SUMMARY:
==25775==    definitely lost: 10,000 bytes in 3 blocks
==25775==    indirectly lost: 0 bytes in 0 blocks
==25775==      possibly lost: 0 bytes in 0 blocks
==25775==    still reachable: 0 bytes in 0 blocks
==25775==         suppressed: 0 bytes in 0 blocks
==25775== Rerun with --leak-check=full to see details of leaked memory
==25775==
==25775== For counts of detected and suppressed errors, rerun with: -v
==25775== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

So probably a good idea to run valgrind first and make it happy before we go into figuring out where heap blocks are going with Massif. Okay, what to do with the information from Massif, anyway? It should be pretty easy to act upon this information. Start with the peak snapshot (worst case scenario) and see where that takes you (if anywhere). You can probably identify some cases where memory is hanging around unnecessarily.

Things to watch out for:

- memory usage climbing over a long period of time, perhaps slowly, but never really decreasing—memory is filling up somehow with some junk?
- large spikes in the graph—why so much allocation and deallocation in a short period?

Other cool things we can do with Massif [**?**]:

- Look into stack allocation (`--stacks=yes`) option. This slows stuff down a lot, and not really necessary since we want to look at heap.
- Look at the children of a process (anything split off with `fork`) if desired.
- Check low level stuff: if we're doing something other than `malloc`, `calloc`, `new`, etc. and doing low level stuff like `mmap` or `brk` that is usually missed, but we can do profiling at page level (`--pages-as-heap=yes`).

As is often the case, we have examined how the tool works on a trivial program. As a live demo, let's see what happens when we take the program complexity up a little bit by (1) looking at the search program we saw in the earlier talk about valgrind; and (2) looking at the original (unmodified) paster.c file from assignment 1 (and then perhaps fixing it and going on). Depending on time available, we may look at some more complex programs.

Cachegrind

Cachegrind is another tool in the package and this one is much more performance oriented than the other two tools. Yes, Valgrind's 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 [**?**].

Cachegrind reports data about:

- The First Level Instruction Cache (I1) [L1 Instruction Cache]
- The First Level Data Cache (D1) [L1 Data Cache]
- The Last Level Cache (LL) [L3 Cache].

Unlike for normal Valgrind operation, you probably want to turn optimizations on (`-O2` or perhaps `-O3` in `gcc`). 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 the search example (same one from above), 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:

```
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. Also be sure that you're reasoning about a realistic workload.

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 [**?**]. Of course, to learn more about how Cachegrind actually does what it does and how it runs the simulation, the manual is worth reading. Not that anybody reads manuals anymore…Just give it a shot, when you get stuck, google the problem, click the first stack overflow link result…

28 — Profiling and Scalability

Profiling and Scalability

Following the discussion of profiling, we should take some time to understand performance and scalability. Recall that what we want in scalability is that we can take our software from 1 user to 100 to 10 million. Finishing work faster helps, but it's not the only way. So to scale up, we probably need to do some profiling to find out what's slow, but we have some things we need to worry about when we want to get away from just working on our dev machines and into big numbers of users.

So we should respect the following principles, as outlined in [**?**].

Hardware Principle. Scalability testing is very different from QA testing (as if you actually test your code!) in that you will do development and QA on your local computer 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 if it scales. If you actually want to test for scalability and real world performance, you should be doing it on the machines that are going to run the program in the live environment. Why? Well, low-end systems have very different limiting factors. You might be limited by the 4GB of RAM in your laptop and that would go away in the 64GB of RAM server you're using. 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 possibly simulate. Legal reasons might prevent you from using actual customer data, but you should be trying to use the best approximation of it that you have. If you only generate some test data, you may not accurately represent the way the users are going to behave the system. Your test set run summary reports occasionally... your users might run them every hour.

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 just not the same.

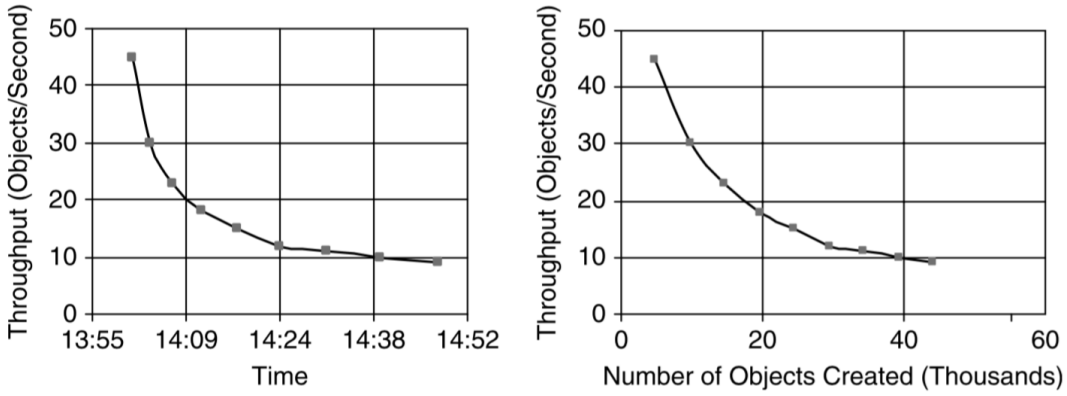
These tests, incidentally, are of great interest to the customers, who would like to know that you can deliver.

Reproducibility and Regression Testing. Your results will need to be reproducible. Remember that good programming practice says that unit tests should be run, and re-run to make sure it all works. The same is true in your performance tests. Just as we don't want to see old bugs cropping up again, old (solved) performance issues are not nice to see either. Or a new change that slows the whole program down is not a success either. So please, have regression testing and performance regression testing.

Characterizing Performance & Scalability Problems

Now that we have some principles, let's get to the application of these principles. It is desirable to characterize problems, in quantitative terms, so that we can solve it. We will use an example from [**?**]. The application is deployed on two systems – one application server and one database server. The data is stored on an external SAN

with RAID0 configuration. A Java[47] program ran to simulate object creation with 15 threads. The performance metric is objects created per second. Let's take a look at the results, also from [**?**]:



This is, to use the technical term, "not good". The throughput deteriorates rapidly from about 45 objects per second down to 9 objects per second, with about 45 000 objects created at the end of the test. This is not the trend we want so see. What if we need to create 500 000 objects? 5 000 000? We have a problem. So we had best find out what's going wrong... and why.

> *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 four potential culprits to start with:

1. CPU
2. Memory
3. Disk
4. Network

These are, obviously, more categories than specific causes, but they are starting points for further investigation. They are listed in some numerical order, but there is no reason why one would have to investigate them in the order defined there.

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 donates its free CPU cycles to world community grid (I'm singlehandedly saving the world, you see.):

```
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. [48] A small guide on how to interpret this, from [**?**].

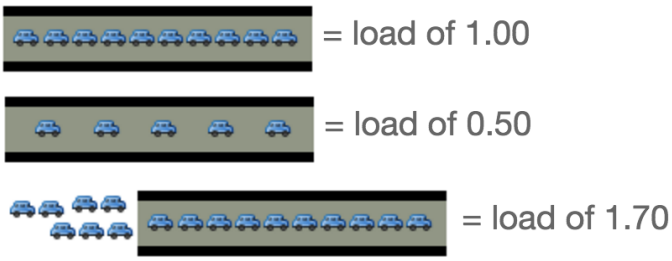
Picture a single core of a CPU as a lane of traffic. You are a bridge operator and so you need to monitor how many cars are waiting to cross that bridge. If no cars are waiting, traffic is good and drivers are happy. If there is a backup of cars, then there will be delays. Our numbering scheme corresponds to this:

[47] Quit snickering... Java isn't THAT slow... anymore...

[48] Why is the uptime so low? I had to shut down and restart the box in September because I moved offices.

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 [**?**]:



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.

Back to our example. Is it CPU? The Application Server CPU utilization, on average, was about 10% and on the database server, about 36%. So that is probably not the cause.

Next on the list is memory. 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 box, 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 though, 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[49].

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:

[49] Yes, a tech journalist named John Dvorak really wrote an article about this, and I will never, ever forgive him for it.

```
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 [**?**]:

```
  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` which requires root privileges[50].

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, back to the original question. The book contains the full story, which is maybe interesting to you if you wanted to dig into the specifics about Oracle 10g and SQL query syntax. I speculate that you do not care about the details, but 96.4% of the total database call time was attributed to database CPU, so let's look at the database queries themselves. Sure enough, the "top 5" queries were taking up a huge amount of time and they all look something like:

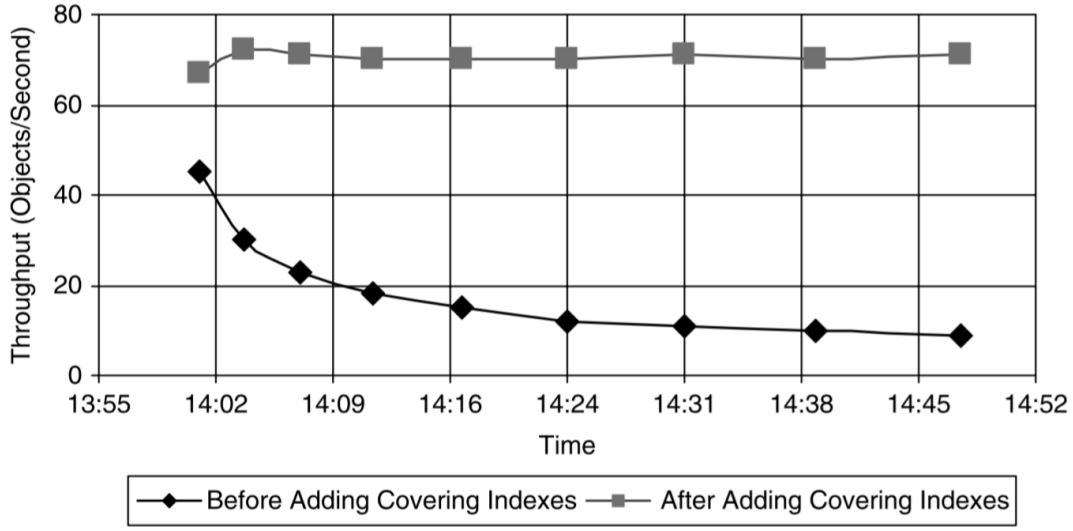
[50] https://xkcd.com/149/

```
SELECT documentId, classId, dataGroupId, consistencyId FROM objectTable WHERE objectID = <value>;
```

Why does performance of this stink? Well, because we're doing a lot of reads from the database. The reads themselves don't necessarily go to disk (cache/buffers/etc may save us here) but we're still doing a lot of reads of data. To speed this up, what we need is to add an additional index for each of these tables.

This is one of the strategies we've talked about before – "be prepared". If we know that the operation like the one above is going to be a common one, then we can tell the SQL server this and have it be prepared for that by defining an index. Some additional work is done to prepare the index and to maintain it when the data in the table is modified, but it makes the query a lot faster when it does come.

And sure enough, we can see the result [**?**]:



29 — 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. If you're lucky, then the problem can be divided among 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.

Message Passing

For the majority of this course, we've talked about shared-memory systems. Last week's discussion of GPU programming moved away from that a bit: we had to explicitly manage copying of data. Message-passing is yet another paradigm. In this paradigm, often we run the same code on a number of nodes. These nodes may potentially run on different computers (a cluster), which communicate over a network.

MPI, the *Message Passing Interface*, is a de facto standard for programming message-passing systems. Communication is explicit in MPI: processes pass data to each other using `MPI_Send` and `MPI_Recv` calls.

Relevant piece about the relevance of MPI today: [**?**]

Hello, World in MPI. As with OpenCL kernels, the first thing to do when writing an MPI program is to figure out what the current process is supposed to compute. Here's fairly standard skeleton code for that, from `http://www.dartmouth.edu/~rc/classes/intro_mpi/`:

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char * argv[])
{
  int rank, size;

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Simple communication example. The slides and live coding example contain a second MPI example which demonstrates `MPI_Send` and `MPI_Recv` usage, also found at `http://en.wikipedia.org/wiki/Message_Passing_Interface`.

Matrix multiplication example. We'll next discuss the code from another MPI example. You can find the code at `http://www.nccs.gov/wp-content/training/mpi-examples/C/matmul.c`. I'll discuss the structure of the code and include relevant excerpts. Here are the steps that the program uses to compute the matrix product AB:

1. Initialize MPI, as in the Hello, World example.

2. If the current process is the master task (task id 0):
   (a) Initialize the matrices.
   (b) Send work to each worker task: row number (offset); number of rows; row contents from A; complete contents of matrix B. For example,

```
MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
```

   (c) Wait for results from all worker tasks (`MPI_Recv`).
   (d) Print results.
3. For all other tasks:
   (a) Receive offset, number of rows, partial matrix A, and complete matrix B, using `MPI_Recv`, e.g.

```
MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
```

   (b) Do the computation.
   (c) Send the results back to the sender.

On communication complexity. To write fast MPI programs, keeping communication complexity down is key. Each step from multicore machines to GPU programming to MPI brings with it an order-of-magnitude decrease in communication bandwidth and a similar increase in latency.

Cloud Computing

Historically, 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.
- 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.

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 [**?**].

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[**?**]. 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

30 — 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 [**?**]:

- 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 [**?**]):

- Server - The banking centre fulfilling customer requests.
- Customer - 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 [**?**]:

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)

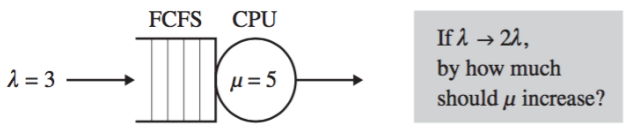
If you're cynical like me, you will think the bank works very hard to not have tellers in the bank to maximize your wait time and minimize their staffing costs. It's actually (allegedly) a trade-off: if I have to wait too long to do my banking, I could always take my business elsewhere (but is that likely 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 [**?**]. 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 $E[T]$.

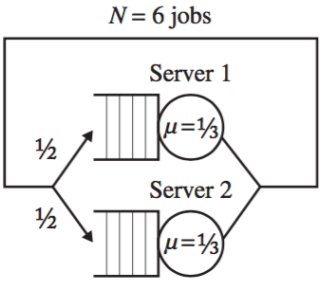
Suppose now that your boss says that tomorrow the arrival rate will double. If you do nothing, you can imagine, there will be a problem: we would have 6 jobs arriving per second, on average, to a system that can service, on average, 5 jobs per second. You have been allocated some budget to replace the 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 [**?**]:



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?

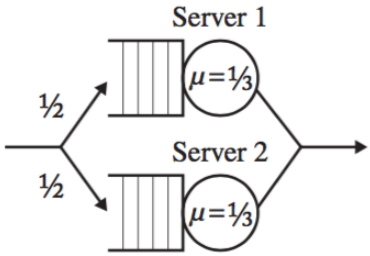
The answer is (3): we don't need to double the CPU speed. We can see later in formal terms why this is the case, but think for a minute about why it is? If we double the service rate and double the arrival rate, we actually get half the mean response time...

Example 2. Okay, how about another example from [**?**]. 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 [**?**]:



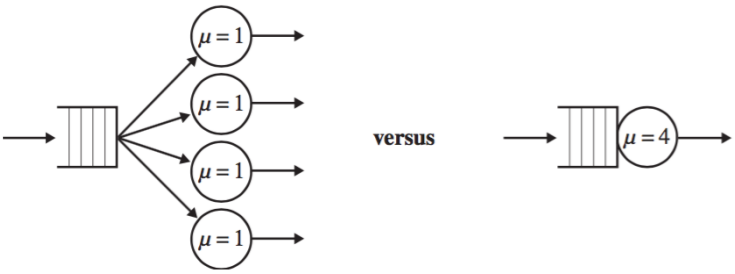
Bad news: sometimes, improvements do nothing. If we replace server one which is twice as fast (so 2 jobs per 3 seconds), does that help? Nope. Not really. Does raising N help? Nope, negligible effect. The bottleneck device is the limiting factor. Strangely, dropping N to 1 means the server replacement 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 [**?**]?



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) [**?**]:



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[51] and

[51] Not less. Fewer. It is countable; therefore fewer. Yes, I am obsessive about this.

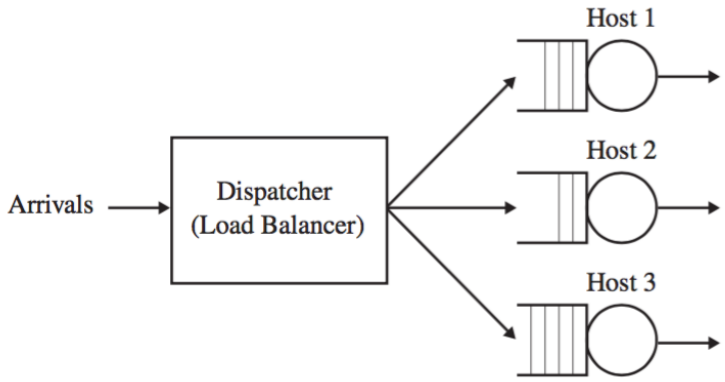
there's one guy who's buying 85 items. You don't want to be standing in line with milk and eggs behind someone who is trying to buy six of everything, do you? So if there's high variability, you probably want multiple servers – the guy buying the whole store can hold up line #1 and you can go to line #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 [**?**]:



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 [**?**]:

- 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[52]

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

[52] 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 [**?**]: 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:

$$E[N(t)] = E[A(t)] - E[D(t)] \geq \lambda t - \mu t = t(\lambda - \mu)$$

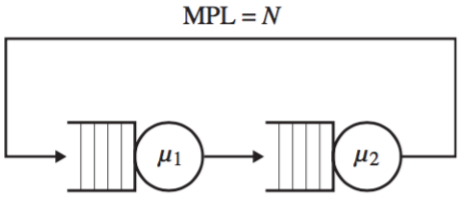
The tiniest bit of calculus says that if arrivals exceed departures, taking the limit as t goes to infinity means $t(\lambda - \mu)$ also goes to infinity. Whoops. So to prevent this terrible situation we just happily assume that this doesn't happen[53].

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 [**?**]:



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 [**?**] 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

[53] 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, depending on whether it is an open or closed system for getting a value for μ, 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

31 — Probability, Convergence, & Ergodicity

A Comically Short Review of Probability

So as a quick preface – to understand some of this material you would be well served to have some familiarity with this subject. I imagine you took a course on probability earlier, but perhaps this material is not exactly fresh in your mind. That being the case, a review of the subject by reading, say [**?**, **?**] will help. But let's go over some of it, in brief.

Suppose we are doing an experiment like rolling dice (a favourite of statistics profs everywhere). We associate the outcomes of these experiments with is called a random variable. A random variable X has many instances, each with some probability; if you roll a fair die X can have instances of $1, 2, 3, 4, 5, 6$, each with a probability of $1/6$ (because we defined this die to be perfectly fair). In a die roll, the events (values of the roll) are discrete - you can't roll 2.1 or 5.5. Time is generally modelled as continuous, but in reality there are limits to how accurately we can measure time on a computer. The resolution depends on the hardware.

Building on this, we can think of the outcome of an experiment as an "event". Probability theory loves to talk about events. We can ask ourselves all kinds of questions like: what is the probability of rolling four 1s on a 20-sided-die in a row[54]? Looking at this from the perspective of application programmers, we say the number of users at time t can be represented by a random variable U and then we can ask ourselves what is the probability of U being greater than 1000?

Probability is usually defined as some sort of experiment with a sample space Ω and some subset of this E is an *event*. A standard example: you have a perfectly fair coin. Possible outcomes are heads, tails, and very rarely, edge (look, it could happen). Each outcome is an event with a certain probability – i.e., chance of happening – usually written $P\{E_n\}$.

If the coin coming up heads is an event E_1 and it coming up tails E_2 are these independent? No, they are not. They are mutually exclusive - if the coin comes up heads it cannot possibly also come up tails. Events are sets and we can do unions and intersections and such. The formal definition of mutual exclusivity is $E_1 \cap E_2 = \emptyset$. Independence, on the other hand, means the probability of E_1 does not change if event E_2 has occurred.

Conditional probability is also important: what is the probability of event E given that event F has occurred? The notation is $P\{E \mid F\}$ and it has a definition (as long as $P\{F\}$ is not zero):

$$P\{E \mid F\} = \frac{P\{E \cap F\}}{P\{F\}}$$

If events E and F are independent, then $P\{E \cap F\}$ is equal to $P\{E\} \cdot P\{F\}$. That is, the probability of E is not affected by the occurrence of F. Suppose you flip a coin four times and it comes up heads each time (event F). What is the probability that the next coin flip will be tails (event E)? Sadly, a lot of people think that after four heads, now tails is "due" and the chance of it being tails is higher than the usual 50-50 (minus the edge case). This is not true – the next coin flip is independent of all the coin flips that went before it.

[54] I played a dwarf warrior in D&D, although truthfully Shadowrun was always much more my style. Elven decker!

A statistics professor would probably come knock down my door and give me a savage beating if I also did not mention Bayes Law. If we want $P\{F \mid E\}$, but have $P\{E \mid F\}$ can we get it? Yes, if we know $P\{F\}$ and $P\{E\}$. Bayes law is:

$$P\{F \mid E\} = \frac{P\{E \mid F\} \cdot P\{F\}}{P\{E\}}$$

For now let us focus on discrete random variables (i.e., those with a countable number of values). For a random variable X we can define a probability mass function (p.m.f.);

$$p_x(a) = P\{X = a\}, \text{ where } \sum_x p_x(x) = 1$$

and the cumulative distribution function is:

$$F_x(a) = P\{X \leq a\} = \sum_{x \leq a} p_x(x)$$

There are five distributions we want to discuss: Bernoulli, Binomial, Geometric, Poisson, and Exponential. The Poisson distribution is the one to focus on, but let's not overlook anything important.

Bernoulli represents a coin flip: so the coin has a probability p of coming up heads and $1 - p$ of coming up tails. So the random variable X associated evaluates to 1 (heads) with probability p and 0 (tails) with probability $1-p$. Thus the p.m.f. is $p_x(1) = p$ and $p_x(0) = 1 - p$.

The Binomial distribution builds upon the Bernoulli distribution. A coin with probability p of coming up heads flipped n times (assuming coin flips are independent). The random variable X if it's Binomial, represents the number of heads when flipping a Bernoulli coin n times. So X can take on values of $\{0, 1, \ldots, n\}$.

The Geometric distribution builds on Bernoulli. Again suppose we have the coin with probability p of coming up heads. If we want to ask how many (independent) coin flips it will take until it comes up heads, the Geometric distribution answers this question. X is the number of flips until we get the result we want.

Applying this to the realm of computing, as in [**?**]. Suppose we have a server farm that has n disks, each of which independently dies (fails) with probability p in a year. So what is the appropriate model for each of these questions?

1. How many disks die in the first year?
2. Given disk d, how long until this disk dies?
3. After one year, is a specific disk d alive or dead?

The answers are (1) Binomial, (2) Geometric, and (3) Bernoulli.

Now let us turn to the Poisson distribution; this is very common in computing. This is our somewhat "natural" idea of randomness. This distribution arises when we have a mixture of a very large number of sources, each with a very small probability. So this is a lot like the number of arrivals to a website or packets at a router in some unit time. If a random variable X has a Poisson distribution, the p.m.f. is:

$$p_x(i) = \frac{e^{-\lambda}\lambda^i}{i!}, \text{ where } i = 0, 1, 2 \ldots.$$

One more thing: there are also continuous random variables, and although we won't talk much about them, we should also consider the exponential distribution. This is a distribution where the probability density function

(p.m.f.) drops off exponentially. If $x \geq 0$, then $f_x(x) = \lambda e^{-\lambda x})$. The cumulative distribution function, i.e., $F_x(x) = P\{X \leq x\}$ is the integration of this function from negative infinity to x, equal to $1 - e^{-\lambda x}$ (again, for $x \geq 0$). What we care about here is actually the inverse of this, $\overline{F_x}(x)$, the probability that the value is greater than x – namely, the outliers or the "tail" of the distribution. Note that we see a constant drop-off of $e^{-\lambda}$ with each unit increase of x. Heavy tails are not very nice.

How about expectation and variance. Recall the notation here $E[X]$ is the expected value of a random variable X. Expected value is, not kidding, just a "weighted average"[55]. Let's say you're entering into a game where you pay $3 to play. If you have a 50% chance of losing all your money, a 10% chance of winning $10, and a 40% chance of getting your $3 back, should you play?

Do the math: your expected return is $(0.5 \times 0) + (0.1 \times 10) + (0.4 \times 3) \rightarrow 0 + 1 + 1.2 = 2.2$. So you expect that by playing this game, for every $3 entry, you get back $2.20 (on average). It doesn't seem like a good investment, does it? Econ 101 logic would tell you the answer is: how much do you value the fun of playing and the hope and excitement of winning? If it's equal to or more than $0.80 then it is worth it.

Proofs of these exist in the textbook(s) but let's not belabour the point by proving it. Just take my word for it that the expectation for a Geometric distribution with probability p is $1/p$: so if we have a coin where the probability of landing heads is $1/3$, then we expect it takes 3 flips to get heads. If a Poisson distribution applies to something, the expectation $E[\lambda] = \lambda$. Okay, that's nice.

The weighted average is nice, but it's not the whole story. If your numbers are 5, 6, and 7, then their average is 6. But the average of the numbers 0, 6, and 12 is also 6. But we should know already that averages can be misleading. A student who takes six courses in a term is unlikely to find that these six exams are distributed equally over the three week exam period as 2, 2, and 2, with at least a little time between any two exams. On the contrary, said student may find there are four in the first week, then a long pause before the next, or two back to back, or even two at once. This student might think the registrar's office hates him or her, but rest assured, it is nothing personal. An average arrival rate of 1 request per second could still, with some probability, produce 3 arrivals in a single second. These should illustrate the idea that we can have an average that seems benign but have outlier situations that are actually messy to deal with.

The formal definition of variance of a random variable X is the expected squared difference of X from its mean:

$$\text{Var}(X) = E[X^2] - (E[X])^2$$

Just take my word for it at the moment that the following table, copied from the book [**?**] is correct:

Distribution	**Mean**	**Variance**
Bernoulli	p	$p(1-p)$
Binomial	np	$np(1-p)$
Geometric	$\frac{1}{p}$	$\frac{1-p}{p^2}$
Poisson	λ	λ

There is plenty more that you should know about (conditional probability, exponential distributions, etc.) but let us break off this thrilling review of probability and put it to use.

Convergence

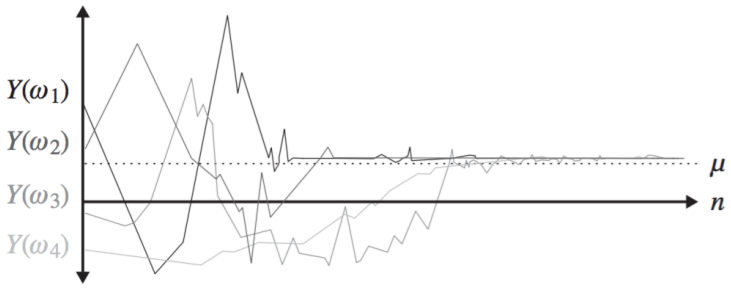
Think back to calculus class. Remember when we talk about limits: $\lim_{x \to \infty} f(x)$. There is an answer for this if the function does (somehow) converge on some value, such as the function $f(x) = 1/x^2$ converging to 0.

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

[55] One of the biggest light bulb moments of my undergraduate life.

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 uncountably many "bad paths", each with probability zero (but that's okay). Zero probability doesn't mean it can't happen, mind you.

An image of what convergence looks like [**?**]:



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

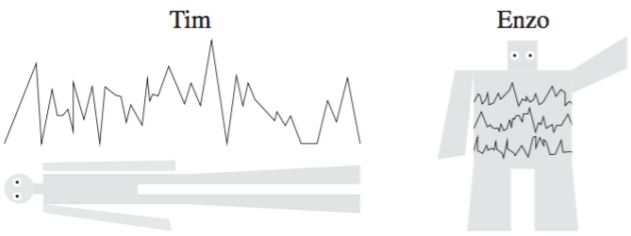
A small but important digression on the subject of sampling, measurement, and testing, from [**?**]. You have an idea of what an average is, but there are two different relevant types of average here—the time average and ensemble average.

Let us just focus on having a single First-Come-First-Serve queue. Every second, a new job arrives with probability p and 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 [**?**]:



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.

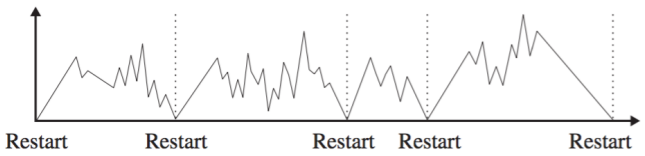
Irreducibility means a process should be able to get from one state to any other state (where state is the number of jobs in the system). This means the initial state of the system does not matter. So if we started at 0 jobs or 10 we could still get to any state in the system (jobs at 2 or 27)...

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, i.e., $t = 1000$, is potentially going to skew the result.

A graphical illustration, also from [**?**], 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 the time in the system of arrival i. The average is taken over one sample path.

$$\text{Ensemble Average} = \lim_{t\to\infty} E[T_i],$$

where $E[T_i]$ is the average time in the system of job i, average being taken over all sample paths.

32 — Applying Queueing Theory

The purpose of the examination in the recent lecture of probability is so that we could able to answer some interesting "what if" questions. So let's take a look at some of those, but first we'll stop to discuss Little's Law.

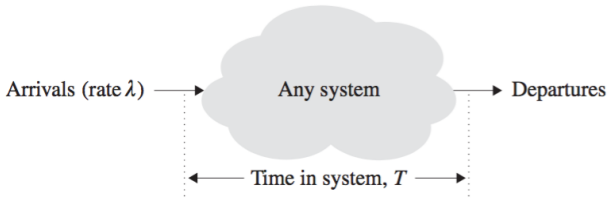
Little's Law

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 [**?**].

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 [**?**]. The basic setup of Little's Law looks something like this [**?**]:



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 [**?**]:

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 $\frac{\rho^2}{1-\rho}$.

An example from [**?**]: 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 $0.01s$, 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 Q 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. 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 [**?**] 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 2.57 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 [**?**]. 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 [**?**] 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 — Practical Scaling: Amazon AWS

Scaling with AWS

After all this discussion about scalability, let's take a look at a talk on the subject of how to scale up using Amazon Web Services (AWS) to your first 11 Million Users. Yes, this one goes to 11.

`https://www.youtube.com/watch?v=vg5onp8TU6Q`

See also [**?**] for a written summary about it.

34 — DevOps for P4P

Two topics today: 1) DevOps considerations (think big); 2) the cost of scalability (think small).

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. Our assignments have been like that, for instance. But a lot of the time we want to keep systems running over time. 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.

The theme today 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 MPI and 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.

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 guys to manage these servers, are you? So, there's DevOps.

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.

Thanks to Chris Jones and Niall Murphy for the following points.

Configuration as code

Systems have long come with complicated configuration options. Sendmail is particularly notorious, but apache and nginx aren't super easy to configure either.[56] The first principle is to treat *configuration as code*. Therefore:

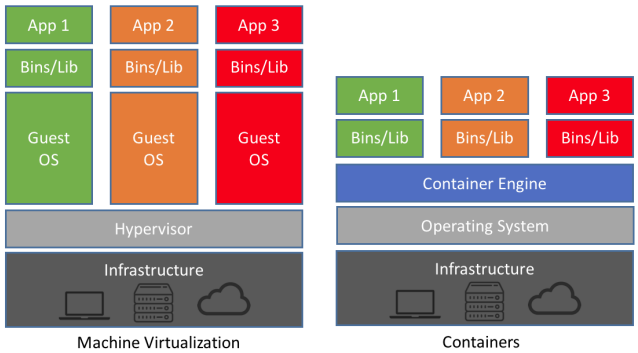
- use version control on your configuration.
- implement code reviews on changes to the configuration.
- test your configurations: that means that you check that they generate expected files, or that they spawn expected services. (Behaviours, or outcomes.) Also, configurations should "converge". Unlike 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 (more on that later).

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). There are tons of great tools like the Red Hat Package Manager (RPM) which will allow you to make the installation and update process automatic and simple. Complicated means mistakes... people forget steps. They are human. Don't let them make mistakes: make it automatic.

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 [**?**]:



[56] If anyone should be foolish enough to want to look into procmail, well, good luck to you...

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.

Common infrastructure

Use APIs to access your infrastructure. That is, you should view different parts of your infrastructure as having an interface and communication is done exclusively via the interface/API. This reduces the coupling between different components, and, as we've discussed, allows you to scale the parts that need scaling.

Try to avoid not-invented-here syndrome: it is usually better to use an open-source tool (or a commercial one) than to roll your own. Some examples might be:

- storage: some sort of access layer (e.g., MongoDB or S3);
- naming and discovery infrastructure (e.g., Consul) (more below);
- monitoring infrastructure (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. PS: a breach of data protection regulations can get very expensive. See `https://www.enforcementtracker.com/` to find out which companies have recently gotten their wrists slapped.

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).

Design for 10× growth, redesign before 100×

[original credit: Jeff Dean at Google] This discussion is based on Martin Fowler's piece on sacrificial architecture: `http://martinfowler.com/bliki/SacrificialArchitecture.html`.

Consider eBay: in 1995, perl scripts; in 1997, C++/Windows; in 2002, Java. Each of these architectures was appropriate at the time, but not as the requirements change. The more sophisticated successor architectures, however, would have been overkill at an earlier time. And it's hard to predict what would be needed in the future.

> "Perf is a feature".
> — Jeff Atwood

That is, you apply developer time to perf, and you make engineering tradeoffs to get it. Some thoughts:

- design with the eventual replacement in mind;
- don't abandon internal quality (e.g. modularity);
- sacrifice individual modules at a time, not the whole system;
- you can also implement new features with a rough draft and deploy to a test audience.

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. We'll talk about systems/VMs[57], but naming is necessary for resources of all kinds.

In brief:

- use canonical one-word names for servers;
- but, use aliases to specify functions, e.g. 1) geography (nyc); 2) environment (dev/tst/stg/prod); 3) purpose (app/sql/etc); and 4) serial number.

This enables you to have a way of referring to each machine in an absolute sense, but also allows you to use functional names when creating dependencies between systems.

There's also the Java package approach of infinite dots: live.application.customer.webdomain.com or however you want to call it. But pick something and be consistent.

Other Topics

Beyond the five principles above, there are a couple more techniques that particularly apply to DevOps:

Continuous Integration. This is now a best practice. It's enabled by the 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 or instant messenger (because e-mail is for old people, right?).[58]

CI is good for all code, but it's especially good for configuration-as-code, which is especially likely to break in different environments.

[57] `http://mnx.io/blog/a-proper-naming-scheme`

[58] I did work at a company where the person who broke the build got a sign outside his cubicle that said IOTD - Idiot of the Day. I'm not too proud to admit that I won this award on my last day of the co-op term.

Canarying. Deploy new software incrementally alongside production software, also known as "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. 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.

Monitoring. Monitoring is surprisingly difficult. There are a lot of recommendations about what to monitor and what to do about it. We care about performance so here are a few things to think about:

- CPU Load
- Memory Utilization
- Disk Space
- Disk I/O
- Network Traffic
- Clock Skew
- Application Response Times

With multiple systems, you will want some sort of dashboard that gives an overview of all the multiple 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. Then you do not necessarily 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.

Here's one way to think about it.

- **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.

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[59] has pulled the fire alarm for a stupid prank or to get out of failing a midterm." This is because we have been trained by far too many false alarms to 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.

[59] This is the PG-13 version of what I actually think.

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 ticket 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 (... grant me some leeway here), will probably take steps to avoid their pain[60]. and they will take steps that keep these things from happening in the future: good processes and monitoring and all that goes with it.

[60] There is a great quotation to this effect by Frédéric Bastiat about how men will avoid pain and work is pain.

35 — Rust

This, Too, Shall Fade and Pass Away

In ECE 459 we've used C and C++ as systems languages. A lot of your previous courses have been in one of those and it's entirely 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.[61]" (17 July 2017) (Holds a PhD in CS from Carnegie Mellon University; was Distinguished Engineer at Mozilla for 10 years; etc.)

What about use of better tools and best practices? March 2019: disclosure of Chrome use-after-free vulnerability[62]; 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 is to make mistakes difficult-to-impossible.

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 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 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.

This brings us to Rust. It is an alternative to C/C++. 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. We'll consider both concepts, but we won't dwell too much on the syntax (mostly for time reasons).

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

It's worth reading up on the topic (outside of lecture) if you are curious about the language, though, and it might help you to understand the examples better.

Rust

This material is based on *The Rust Programming Language* by Steve Klabnik and Carol Nichols [**?**] and I'll make references as appropriate.

Here's some Rust code.

```
fn main() {
  let x = 42; // NB: Rust infers type "s32" for x.
  println!("x is {}", x);
}
```

By default, Rust variables are *immutable*.

```
fn main() {
  let x = 42; // NB: Rust infers type "s32" for x.
  x = 17; // compile-time error!
}
```

Let's consider two examples that look similar but have drastically different meanings.

```
let x = 1729;
let x = 88;
println!("shadowed x is {}", x);
```

```
let mut x = 33; // mutable
x = 5;
println!("mutated x is {}", x);
```

In the first case, old "x" still exists but is inaccessible under the name "x". In the second case, the storage cell for "x" used to contain 33 and then contains 5. The difference matters, for instance, when there are references to "x". This example is a bit silly though; the real usage for shadowing is perhaps something more familiar, specifically parsing (or other transformation):

```
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. We like this because we can re-use the variable name without having things like `guess` and `guess_parsed` or other "what do I call this now" problems.

Rust immutability. By default, a variable in Rust is immutable. You can make it mutable if you choose, explicitly by declaring it as mutable. Lots of concurrency issues involve the internal state of objects that are accessed by different threads. Structs or tuples are either all mutable or all immutable. (Although interior mutability is a thing in Rust. We're not talking about it.)

Rust obviously has compile-time constants and they are truly unmodifiable. These have to be known at compile time, and are truly a fixed value. This is different from an immutable type which is determined at runtime but cannot be changed once it has been assigned.

In C, you can cast away `const`-ness; not so in Rust. If something is not mutable in Rust, you can't cast it into mutability.

Perf implications. We mentioned immutability in Lecture 7. The best way to avoid having to use locks: have no writes. (Even read/write locks require writes to acquire the read lock). However, there's a tradeoff. If your data structure is immutable but you want to update it (as we often do with data structures), you need to copy the data structure, at least partially. That can be slow.

Runtime safety. We said that Rust is safe. One way in which it is safe is for arrays. Rust has tuples and structs. Hard to go out of bounds on those. Rust also has arrays. Like with any language, one can imagine going beyond the ends of an array. Rust defines the behaviour of going beyond the end of an array: it is a runtime exception ("panic"), unlike C/C++, where it is undefined behaviour (anything can happen).

```
let a = [1,2,3,4,5];
let index = 10;
println!("error!␣{}", a[index]); // panics here.
```

What's special about Rust?

Let's step back and do some Rust propaganda.

- harder to write unsafe code: compiler + runtime ensure safety. No arrays-out-of-bounds accesses, null pointers (at all), wild pointers;
- yet can still write low-level code;
- supports zero-cost abstractions (like C++);
- designed with ergonomics in mind;
- type system obviates need for either garbage collection or manual memory management[63] (and you'll get manual memory management wrong);
- type system prevents race conditions;
- dependency management using crates.

As far as I know, Firefox's rendering engine, Project Servo, is the largest deployed Rust codebase, at 350 kLOC of non-test code in January 2020.

Ownership in Rust [Chapter 4.1]

Also known as "how to fight the borrow checker and win".

Rust uses ownership types to manage its heap. Ownership types were not invented by the Rust community, but Rust is the first production-scale language to deploy it. The alternatives are `malloc/free` in C, or `new`/GC in Java. Rust does still have a stack, but we'll see when things go on the stack vs when they go on the heap.

The Rules

1. Each value in Rust has a variable that *owns* it.
2. This variable is *unique*.
3. When the owner goes out of scope, the value will be dropped (aka freed).

Variable scopes are fairly standard.

```
fn main() {
  println!("start");
  { // no s
    let s = "I␣am␣s";
    println!("s␣is␣{}", s);
  } // s now out of scope
}
```

OK, let's put something on the heap. We'll be using Rust `String` objects rather than string literals. String literals are compile-time constants. String objects contain a heap component, which may be allocated and freed.

(What can go wrong with heap allocation? You might not free/free too late; free too early; double free. GC manages this through an approximation: if you have no more pointers to it, then it doesn't spark joy, and you don't need it anymore. For Rust, this is not the way.)

[63] Well, mostly. Sometimes you need to use ref-counted data, and we'll see that.

```
fn main() {
  let s = String::from("hello"); // immutable String
  let mut s2 = String::from("459 assignments"); // mutable String
  s2.push_str(", maybe?");
  println!("got string {}", s);
}
```

Rust uses rule #3: if something goes out of scope, then drop (free) it. This is quite like C++ RAII (Resource Acquisition is Initialization).

Still, we need a solution for objects that live beyond their original scope, e.g. return values.

```
fn return_a_string() -> String {
  let s = String::from("longevity");
  return s; // transfers ownership (moves) to caller
}

fn main() {
  let returned_string = return_a_string();
  println!("string {}", returned_string);
}
```

So, Rust frees owned values when variables go out of scope. Also, Rust calls "drop" (akin to a destructor) on objects that go out of scope. Note that going out of scope, not the drop call, is what actually causes the free.

Transferring Ownership (aka move semantics)

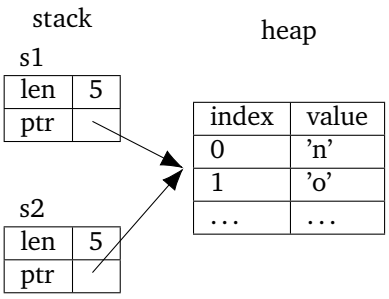
Let's look at an example.

```
let s1 = String::from("no surprise");
println!("can print {}", s1);
let s2 = s1;
println!("can still print {}", s2);
println!("this line won't compile! {}", s1); // no longer owns
```

(Note that string literals, or ints, or anything only on the stack, doesn't have this behaviour—they are copied, or technically, they have the "Copy" trait.)

OK, so what's going on? Let's take a step back.

Rust strings are a hybrid, containing both a stack part and a heap part.



The assignment `let s2 = s1` carries out a shallow copy of the stack part. Rust does not automatically copy the heap part.

Now, recall that Rust has automatic memory reclamation. How can that work? What gets freed? Here, `s1` and `s2` are in the same scope. When they go out of scope, what should get freed? We don't want to double free.

Key idea. For automatic memory reclamation to work, we give `let s2 = s1` *move semantics* (as in C++). After the move, `s1` is no longer valid. Ownership of the heap part is moved from `s1` to `s2` by the assignment.

That is, `s1` no longer owns the heap object and is not responsible for freeing the heap part when it goes out of scope. Only when `s2` goes out of scope do we free the heap object. And because the heap object only has one owner, it is only freed once.

Note: deep copy is possible with "clone", but we have to trigger that explicitly.

Want to know more about ownership? Here's a blog post:

`http://squidarth.com/rc/rust/2018/05/31/rust-borrowing-and-ownership.html`

Functions and Return Values

Move semantics also applies to function calls and return values, e.g.

```
fn main() {
  let s = String::from("moving␣to␣callee");
  callee(s); // afterwards, s is invalid
}
fn callee(param:String) {
  println!("got␣{}", param);
} // param goes out of scope, object dropped
```

If you return something, then the ownership passes back to the caller. Tuples help pass ownership of multiple objects, but this is still quite high-overhead for developers.

```
fn main() {
  let s = String::from("459");
  let len = calculate_length(s);
  println!("string␣{}␣has␣length␣{}", s /* we still have it! */,
           len);
}
fn calculate_length(s:&String) -> usize { // note the & for borrow
  s.len() // last expr is return value
} // s is ref so nothing goes out of scope
```

Borrowing and mutation

"Can I please borrow your object?" Like other variables, references are immutable by default. We can have mutable references, though.

```
fn change(s:&mut String) {
  s.push_str("more");
}
fn main() {
  let mut main_str = String::from("some␣");
  change(&mut main_str); // create mutable ref to main_str
}
```

There can be only one (mutable). The following code won't compile:

```
let mut s = String::from("one");
let r1 = &mut s;
let r2 = &mut s; // rustc complains!
r1.push_str("two");
```

In fact, while `r1` is in scope and a future use of `r1` is to execute[64], you can't do anything with the original `s`. The only way to access the string is through `r1`. After the last use of `r1`, you can create `r2`.

Since there is only one way to access `r1`, then there will be no race conditions.

This is OK:

```
let mut s = String::from("one");
let r1 = &s;
let r2 = &s; // no problem!
```

But you can't then do `let r3 = &mut s;`.

How many? You can have as many outstanding immutable refs as you want. If there are any immutable refs, you can't have *any* mutable refs. The mutable ref can't be created until the last use of the immutable ref.

You also can't commit use-after-free errors: you can't return a ref that outlives its value.

```
fn dangle() -> &String {
  let s = String::from("hello");
  &s // rustc complains: s goes out of scope with active refs
}
```

Rust also has *smart pointers*, which may be reference counted. This is like C++'s smart pointers, specifically `shared_ptr` (but Rust can tell you about some things at compile time which C++ will tell you at runtime). Normal Rust objects are more like `unique_ptr`.

We need reference counted heap objects e.g. to implement graphs. We don't have enough time to talk about smart pointers, but Chapter 15 of the Rust book is good.

Fearless Concurrency

As with many other aspects of Rust, we trade compiler errors for runtime errors; in this case, runtime concurrency errors like race conditions. That is, the type system ensures concurrency safety!

Rust uses a fork/join model like pthreads. It delegates to the operating system's threads support and hence implements 1:1 threads.

```
let handle = thread::spawn(|| { // closure (can put args between ||)
  // thread code goes here
});
// main thread continues here
handle.join.unwrap(); // unwrap: panic in case of error
```

This is not too different from C++.

OK, how do we share data between threads? We can move it from main to thread:

```
let v = vec![1,2,3];
let handle = thread::spawn(move || { // move: everything accessed inside closure is moved
  println!("vector␣{:?}", v);
}); // no longer have access to v in main
handle.join().unwrap();
```

Rust is saving you from being able to concurrently access `v` in main and thread.

But that's only one way! This isn't quite enough.

[64] The borrow checker got upgraded in 2018 to support Non-Lexical Lifetimes. It's kind of complicated, but see `http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/` and related posts for more information.

Message Passing

One way to share data is message passing. We've seen this before (OpenMPI). In this case, each value still only has one owner. We use *channels*. The ownership passes from the sender, through the channel, to the receiver.

```
use std::thread;
use std::sync::mpsc; // multi producer, single consumer

fn main() {
  let (tx, rx) = mpsc::channel(); // tx is cloneable
  thread::spawn(move || { // here, tx goes to closure
    let val = String::from("april");
    tx.send(val).unwrap(); // val moved from sender
  });
  let received = rx.recv().unwrap();
  println!("got:␣{}", received);
}
```

Note the send/receive pair. There is also `try_recv` to do nonblocking receives.

Shared State

People debated for a long time which was better: shared state (like pthreads) or channels. Rust supports both. Of course, the problem with shared state is race conditions. Like manual memory management, we can manually acquire and release mutexes. What could possibly go wrong? Rust's ownership system will help.

We'll need to talk about multiple ownership. But let's talk about mutexes first.

```
use std::sync::Mutex;
fn main() {
  let m = Mutex::new(5); // mutex guards access to an i32
  {
    let mut num = m.lock().unwrap();
    // unwrap: maybe some other thread panicked while holding lock;
    //         then we panic too.
    *num = 6; // "deref" the mutex (is actually a smart pointer)
  } // release lock when num goes out of scope
  println!("m={:?}", m);
}
```

Well, that's fine, but it's just one thread. We really do need multiple ownership to share data. The shared data needs to be owned by all threads, and a naive solution will get rejected by the borrow checker. Instead, we have to use *reference counted cells*, implemented by `Arc`.

```
use std::sync::{Mutex, Arc};
use std::thread;
fn main() {
  let counter = Arc::new(Mutex::new(0)); // atomic reference cell
  let mut handles = vec![];

  for _ in 0..10 {
    let counter = Arc::clone(&counter); // clone the Arc
    let handle = thread::spawn(move || {
      let mut num = counter.lock().unwrap();
      *num += 1;
    });
    handles.push(handle);
  }
  for handle in handles {
    handle.join().unwrap();
  }
  println!("result:␣{}", *counter.lock().unwrap());
}
```

Rust guarantees that you have the appropriate lock, using ownership (possibly multiple ownership). Rust does not guarantee lack of deadlocks.

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.

[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.

[Bed17] M. Bedford Taylor. The evolution of Bitcoin hardware. *Computer*, 50(9):58–66, 2017.

[Can06] Bryan Cantrill. Hidden in Plain Sight, 2006. Online; accessed 20-Janaury-2016. URL: http://queue.acm.org/detail.cfm?id=1117401.

[CDF14] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *IEEE Transactions on Software Engineering*, 40(12):1185–1205, 2014.

[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): Whats 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.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association. URL: http://portal.acm.org/citation.cfm?id=1247415.1247417.

[Dev15a] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler, 2015. Online; accessed 25-November-2015. URL: http://valgrind.org/docs/manual/cg-manual.html.

[Dev15b] Valgrind Developers. Helgrind: a thread error detector, 2015. Online; accessed 25-November-2015. URL: http://valgrind.org/docs/manual/hg-manual.html.

[Dev16] Valgrind Developers. Massif: a heap profiler, 2016. Online; accessed 23-January-2016. URL: http://valgrind.org/docs/manual/ms-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/.

[Duf10] Joe Duffy. The 'premature optimization is evil' myth, 2010. Online; accessed 2-January-2017. URL: http://joeduffyblog.com/2010/09/06/the-premature-optimization-is-evil-myth/.

[Duf16] Joe Duffy. Performance culture, 2016. Online; accessed 28-December-2016. URL: http://joeduffyblog.com/2016/04/10/performance-culture/.

[Dur15] Jonathan Dursi. HPC is dying, and MPI is killing it, 2015. Online; accessed 6-January-2016. URL: http://www.dursi.ca/hpc-is-dying-and-mpi-is-killing-it/.

[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/.

[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.

[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.

[Gro15] Khronos Group. OpenCL 2.1 and SPIR-V 1.0 Launch, 2015. Online; accessed 6-January-2016. URL: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf.

[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.

[Hof16] Todd Hoff. A Beginner's Guide To Scaling To 11 Million+ Users on Amazon's AWS, 2016. Online; accessed 16-January-2016. URL: http://highscalability.com/blog/2016/1/11/a-beginners-guide-to-scaling-to-11-million-users-on-amazons.html.

[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://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f.

[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.

[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.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018. https://doc.rust-lang.org/book/.

[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.

[Kul09] Kestas Kuliukas. How rainbow tables work, 2009. Online; accessed 17-December-2018. URL: http://kestas.kuliukas.com/RainbowTables/.

[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/.

[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.

[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.

[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.

[Ngu17] Nick Nguyen. The best Firefox ever, 2017. Online; accessed 2019-12-18. URL: https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/.

[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/.

[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.

[Pol17] Ryan Pollock. The search for the Goldilocks browser and why Firefox might be "just right" for you, 2017. Online; accessed 2019-12-18. URL: https://medium.com/mozilla-tech/the-search-for-the-goldilocks-browser-and-why-firefox-may-be-just-right-for-you-1f520506a

[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.

[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.

[Tan05] Brian K. Tanaka. Monitoring Virtual Memory with vmstat, 2005. Online; accessed 13-February-2016. URL: http://www.linuxjournal.com/article/8178.

[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.

[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.