

The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high-quality educational resources for free. To make a donation or to view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at [ocw.mit.edu](https://ocw.mit.edu).

**CHARLES E. LEISERSON:**

OK, let's get started. So today we're going to talk about measurement and timing. And I want to start out by just showing you a study that one of my students did-- actually, at that point, he was a former student-- whereas timing a code for sorting. So here's the code. This isn't exactly his code, but it's in the same spirit so that you get the idea.

And so let's just run through this code and take a look to see what it's doing. It's pretty straightforward. We're going to use the `time.h` header file to get access to the `clock_gettime()` routine, which is going to be what we used to get timing measurements. And then we have a sorting routine that we're going to time. That I'm not showing you.

And there is also a fill routine, which is going to fill up the array with numbers-- with random numbers, so we have something to sort. And the `clock_gettime()` uses a struct that is defined here, and so I'm defining two timing structs-- a start and an end. So this is just absolute boilerplate setting up for taking timing measurements. And in this case, basically the high order part of the struct tells the seconds, the lower part tells the nanoseconds.

And then we're going to loop over arrays of increasing length. And then what we're going to do is fill them up-- oh, I forgot the fill-- and then we're going to measure how much time-- what the time is just before we sort. Then we're going to sort, and then we're going to measure the time after sorting. And then we compute the difference, and figure out what the elapsed time is, print that out, and then we do it again for a little bit larger array.

So is that clear what's going on? So we're just sorting a bunch of numbers, then we're sorting some bigger ones, sorting some bigger ones, so we can see what the growth of the sorting routine should be. People have a pretty good understanding of what the code does? OK, so what do we expect to see? What's this curve going to look like? What are some properties of it? Yep?

**AUDIENCE:**

[INAUDIBLE]

**CHARLES E.**

So micro is  $n \log n$ , but it's certainly going to grow, right? It should be up and to the right. In fact, one rule, if you ever get into marketing, is that all graphs must go up and to the right. If they're going down and to the right, then your company's in trouble. So it should be going up and to the right, and it should follow, for example,  $n \log n$ , if it's an  $n \log n$  sort, which is what this one was. I think he was, in this case, timing a merge sort. They should go up and to the right, and should follow  $n \log n$ , or whatever.

**LEISERSON:**

So let's see what actually happened when he took the measurements. This is actually his data from-- gosh, this must have been 20 years ago or something. Here's what it looked like. So the blue Xs there are the runtimes. And then through that, we've plotted two curves, one which is the best fit to order  $n \log n$  growth, and the best fit to order  $n$  growth.

You notice that for-- even though we're going up to 4 million here, there's not that much difference between  $n \log n$  and order  $n$ . You can see it mostly down in the tails. Definitely the  $n \log n$  follows a little bit better, but really,  $\log n$  is pretty small already. But wow, those measured times-- so if you look, there are points way up here-- really slow. It starts out-- it goes slow a little bit, and then it gets a little bit worse, and then a little bit worse, and a little bit worse.

Notice also that the bumps are getting closer and closer together. What is going on? Why? I don't know about you, but I thought the data would follow the green dots reasonably closely. But you can see it doesn't. It's always good to have a model for what you think is going on because then, when you-- because some people will just take numbers. They'll say, here's my numbers for my-- that I've measured.

And if you don't actually have a model for what those numbers mean, you're probably fooling yourself. You're more likely to have made some sort of error, or there's something going on that you're not observing, or whatever, if you don't actually have a model for what you think should be going on. So what's going on here? Who can suggest a hypothesis for what is going on?

So he took these numbers on his laptop, by the way. What do you suppose is happening here? Some ideas. Yeah?

**AUDIENCE:**

[INAUDIBLE]

**CHARLES E.**

Maybe it doesn't fit in the cache. What would you expect to happen if things didn't fit in the cache?

**LEISERSON:**

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Yeah, you sort of think that it would go along, and then it would jump. So interesting issue, but that doesn't seem to be what's happening there. It's going up, then it's going back down. It's going up and going back down-- roller coaster. What other ideas are there? Good idea. Good idea. Let's think a little bit about what's going on in the machine. What are some other good ideas? Or bad ideas? Let's eliminate some. Yeah.

**AUDIENCE:** They're not powers of 2.

**CHARLES E. LEISERSON:** They're not powers of 2. These are not powers of 2, right? Because they're getting closer and closer as we get bigger and bigger. Yeah, so you're right. It's not correlated with powers of 2. Weird. Because sometimes things are alignment issues, and we'll talk more about that. It will come up when we talk about caching after the quiz. Everybody knows there's a quiz next time-- especially all of you who aren't here?

OK, so what else might be going on in the machine here? Because this is reality. This is what happens when you take measurements. So we're being very nice to you. We're giving you AWS run. We have done everything we can to make sure those numbers come out clean, and beautiful, and untouched. There they are. That is quality measurements we're taking for you. But if you had to do it yourself, that's what this lecture, in part, is about. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** So you think that there may be something having to do with the cache. But I'm going through each time and I'm refilling the array each time, so they're kind of starting from a clean slate-- similar clean slate each time. What else is going on in the machine here? Yeah?

**AUDIENCE:** [INAUDIBLE] totally unrelated stuff [INAUDIBLE]

**CHARLES E. LEISERSON:** Yeah, there could be totally unrelated stuff running. You might have daemons, you might have all kinds of things, and so forth. So he thought of that, and he shut down everything he possibly could. And this is what he got still. But that's a great idea because often, there's some external things going on. In this case, it's called multi-tenancy. There's more than one thing using the computer at a time. Good idea, but happens not to be the one.

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Could be precision with the timing. Yeah, sometimes there can be issues there, but this was not a precision issue. He could have used a really dumb timer and gotten something very similar to this. What else is going on your machine? Yeah?

**AUDIENCE:** Maybe his machine's checking for updates every minute.

**CHARLES E. LEISERSON:** Yeah, maybe it's checking for updates. That's once again some external things. No, it wasn't checking for updates. Wasn't checking for updates. What is going on here? What is going on? Let's have some more ideas. What other things might disrupt measurements? Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Yeah. This was actually merge sort he was timing, so there's no randomization. But even that, if it were quick sort, it'd be at random that things would tend to take longer, rather than following this crazy pattern. What is causing that crazy pattern? Yeah?

**AUDIENCE:** Does the random fill have to do with the time?

**CHARLES E. LEISERSON:** No, because the random fill is done outside the timer. Each time through the loop, we fill, and then we start the timer, and then we take the measurement, and so forth. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** It's not allocating memory. But that's an interesting idea, because sometimes you have things going on where you think things are happening right away, but the system is being clever and delaying it. And so you end up paying for it at some later time, and that could possibly create something. Turns out not to be what's going on here.

So what's happening here is that the machine is changing the clock frequency. Why is the machine changing the clock frequency? Your laptops change the-- the systems that we have, they change clock frequency. Why do they change it?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Because the laptop is getting hot. So what do are doing? We're running something computational. And the smaller ones-- OK, we get a lot of those done, until it starts heating up, and so it slows down the system clock to save power. OK, and then what happens? Slows it down a little bit, cools off a little bit, starts to speed up again. And then we run longer. And why

are these things getting closer and closer together? Yeah?

**AUDIENCE:** Takes longer and longer to run the sorts.

**CHARLES E. LEISERSON:** Yeah, it takes longer and longer to run the sorts, so you're going to see the effect closer in an interval here, even if it happened to be equal in time. Even if it was equal in time, we're doing bigger and bigger problems. This is nuts, right? We want to take measurements. We want to know whether the software is faster. What are you supposed to do?

So here, if you just took a measurement and said, look, this is the time that it takes me to run this code, you would be hugely missing the boat, if you were taking one of those high measurements. You compare A to B. You run A first, then you run B. B is slower. Oh, well, that's because when you ran A, it heated up the processor, so the processor slowed it down.

So this particular architectural feature is called DVFS, dynamic frequency and voltage scaling. It's a technique to reduce power by adjusting the clock frequency and supply voltage to transistors. So the idea is that, if the chip ends up getting too hot, or in the case of laptops, often if you want to conserve battery, it chooses to slow down the clock. And the second thing it can do is reduce the voltage, if the frequency is reduced.

So when you're actually running slower, you can actually get the same reliability of switching with a lower voltage. At a higher voltage-- sorry-- at higher clock frequencies you need enough voltage to make sure those electrons are scooting across the transistor junctions fast enough. So the basic power law that the electrical engineers-- is anybody here an electrical engineer? OK. There's good stuff in EE, let me tell you.

So those of you who are too embarrassed to raise your hands, I support EE. So power goes as  $CV^2f$ , where C is what's called the dynamic capacitance. There's actually another term, which is the static capacitance, which doesn't have to deal with frequency or whatever. But for the dynamic power, it's  $CV^2f$ . It's the dynamic capacitance. It's roughly the area of the circuitry times how many bits are moving. So if bits don't move, they don't consume power, for dynamic power.

And then V is the supply voltage, and then f as the clock frequency. So if you can reduce the frequency and voltage, you get a cubic reduction in power, and also in heat. Who thinks their battery doesn't last long enough? Yeah, OK. Wouldn't be nice if they lasted a month? So you can see why they're motivated to play this game-- to save the battery or to run things as hot as

they can. But if it gets too hot, we'll just back off. But for performance measurement, this is basically a nightmare. It wreaks havoc.

So the topic of today's lecture is how can one reliably measure the performance of software, when you have stuff like this going on in our system? Now, the good thing about something like DVFS is there's actually ways of shutting it off and taking measurements. You probably pay some performance for doing that, because they can basically-- who's ever worked on an overclocked system? Some of you game players, right?

So part of the game-- the idea there, when they're overclocking it, is let's keep-- get things cool and so forth. And there's lots of games like this that are played and overclocked systems. So I'm going to talk about three topics today. One is about quiescing systems, which is making them quiet enough that we can take good measurements, getting rid of some of the noise. And that second I talk about some tools for measuring software performance, and then we'll talk a bit about performance modeling.

So I hope this is a little motivational. Boy, you think you've seen it all sometimes, and then somebody presents you with a puzzle, and it's like, what's going on? And then takes a little detective work to figure out that that's actually what's going on. So we'll start first about quiescing systems. Who knows who Genichi Taguchi was? Anybody here? No? He's very famous because he's the one who made our automobile industry produce reliable cars. Very, very famous fellow.

He did a lot of different things. I'm going to point out one of the things that he observed. And it's in the context of suppose you were an Olympic pistol coach, and you've got two shooters-- A and B-- and you look at their targets. And here's A's target. Oh, he hit the bulls-eye. Good. And if you add that up, he's got a pretty decent score. Then you look at B. B didn't even hit the bullseye, and his score is really lousy. Who do you want on your team, A or B? Who do you want?

**AUDIENCE:** You would pick B, just because he could adjust the sights.

**CHARLES E. LEISERSON:** Yeah, because it's easy to tell him what to do. You adjust the sights. You teach him to shoot down a little bit lower to the left. It's easy to diagnose what his problem is. That's all you have to do. What do you do for A? It's not clear what you do. And so what he did in the realm of quality control was say, look, when we're producing products, before we try to fix the reliability, or whatever is, of the product-- increase the quality of the product-- let's first reliably produce

whatever we're producing so we get the same thing every time. Because then we can go in and see what the problem is, and we can adjust for it.

And so the idea is go after the variance first. Go after the spread. Try to get the spread as small as possible. Try to figure out how it is that you can isolate-- produce the same thing, even if it's not very good. And then go and try to make changes. Because if you try to make changes while you have high variance, you don't even know if you're making progress or not, because it could be in the noise of your measurements.

Now, in the context of performance engineering, if you can reduce variability, you can do things like compensate for systematic and random measurement errors. And you can also do things like not run as many trials to discover whether one program is better than another. So a lot of advantages to being able to have really quiet things. It's kind of obvious that you should want to have reliable measurements, but there's actually, as I say, a theory behind it for wide is that you want to have quiet measurements, and what you need to focus on.

Now, in our computer systems, there are lots of sources of variability, and some people came up with some of them here. Let me mention just a few of them here. So there are things like daemons and background jobs, things that are running on your system that are helping the system to do whatever it's doing. Many of them are unnecessary, and those can be running-- so for example, maybe you set up a cron job to do something for you every night, and that happens to be just when you happen to be making some measurements. Well, that's not really good, because you're now sharing the resource.

Interrupts-- something comes in. I had one time where we were measuring stuff on the connection machine CM5, which in its day, was the world's most powerful computer. 1993, it was the top of the list of the supercomputers. And now this computer is more powerful. Cost \$43 million or something. Moore's law has really made a difference. And we were measuring the performance of something, and we kept getting these anomalous results.

And eventually, we tracked it down. It took us a lot of work, but we tracked it down. The graduate student who was running it, while he was running it, he was moving the mouse around. And when you moved the mouse, it was interrupting 200 times a second to deal with the interrupts that was interfering with our measurements. True story. True story. We disconnected the network even so that we wouldn't get external things, and we're still getting this thing. What is going on? And eventually, we tracked it down to that fact that he was just

sitting there idling-- so it's like, start it up, and then hands off. It was like, nobody move.

Code and data alignment-- where the code lies affects its performance. If a code goes across two cache lines versus completely within one cache line, that can have a difference, depending upon what's conflicting in the cache. Or if you go across page boundaries, it can have a very big difference, for example, in the translation lookaside buffer, TLB. You may be going for different entries. There may be different things there. So code alignment can make a difference in the-- what you're doing.

Thread placement-- if you have a multicore machine, which core is it running? It turns out the system likes to use core 0 for a lot of its stuff. So if you're going to take reliable measurements, don't run on core 0. Runtime scheduler-- the fact that, for example, we have a randomized scheduler, which means that there's random numbers going on so you're going to try to take measurements in the midst of all this randomness.

Hyperthreading-- hyperthreading is where they take one core. It's also called symmetric multithreading-- or no, simultaneous multithreading. That's what it's called, simultaneous multithreading, or hyperthreading is what Intel calls it. What that is is they have one functional unit, and then they run two instruction streams through it at the same time, each with its own set of registers, but using the same functional units. And they get basically a 20% speed up or something from having-- it looks, from the software point of view, like you've got two processors, but really it only gives you 1.2 processors.

So it's one of these things that, by the way, if you start counting up how many processors and saying work over the number of processors, and you say, well, how many processors do I have-- well, you may have just a hyperthreaded processor, rather than a real processor. So one thing, for example, in the cloud system that you folks are using, we turn off hyperthreading so that we can get more reliable measurements, and so we can look at the measurements as a function of the number of processors.

Multitenancy-- and this is particularly important in the cloud. If you're in the cloud, there are other people using the system, they can end up using lots of resources that maybe you want, like cache. And they can end up using network traffic that may affect you, and such. I'm actually quite amazed at how well Amazon does in AWS in making it so that that stuff doesn't affect you very much. Our numbers show that they are, in fact-- they are definitely the leader right now in having repeatable measurements, compared to all the cloud providers.



We talked about DVFS. There's also another one called Turbo Boost. So Turbo Boost looks to see how many jobs are actually running on the multicore. If there's only one job running on the multicore, it increases the clock frequency for that job-- or if there's just a few. As soon as another one comes in, it slows things back down. So it tries to give a boost when you're in executing serial code because it says, well, I'm not generating heat from all the cores. I just have to generate it from the one that I've got. I can afford to have one of them run hotter.

Network traffic-- and there's, by the way, a bunch of other ones. We're going to talk about a few of these, but first let me tell you what the impact of quiescing a system is. So this is joint work that I did in my group with Tim Kaler. So we wrote a Cilk program to count the primes in an interval. And we ran on a c4 instance, 18 cores-- similar to what you're running with. We had two-way hyper-threading on. Turbo Boost was on. We had 18 Cilk workers. We had 100 runs, each about one second.

And what I've plotted here is how much is the percent that each run was. We basically ran 100 runs, and then I sorted them from smallest run to largest. And I normalized them to whatever the minimum one was so that each thing here is a percentage above the minimum. So you can see that the slowest run was almost 25% slower than the fastest run. So you can see what the impact is. So 25%-- if you're trying to produce a code by getting a 3% improvement 30 times, you can't-- there's so much noise here. It's very hard for you to figure out that you're 3% faster.

So if you quiesce the system, same thing. We turn hyperthreading off, turn Turbo Boost off, and we quieted all the daemons, and so forth. Then out of 100 runs, we got essentially the same value every single time, except for three times. And notice that the scale here has changed. The scale here is now less than 0.8%, less than 1% slower. So this says that, hey, if I took a couple of measurements, I'm very likely to hit something that is the real running time, which tends to be, for this, what the minimum is here. Because all the other stuff is noise that's just adding in.

Make sense? So it is possible to quiesce a system. It's not that hard, it just takes a small matter of programming and systems work. So here are some tips on how you quiesce a system, if you wanted to do this for your laptop, for example. Make sure no other jobs are running. Shut down daemons and cron jobs. Disconnect the network. Don't fiddle with the mouse. For serial jobs, don't run on core 0, where interrupt handlers are usually run.

Turn hyperthreading off. Turn off DVFS. Turn off Turbo Boost. Use taskset to pin workers to cores. So taskset is a utility that says, I want you to run this thread on this core, and don't let the operating system bounce it around. So normally, the operating system maps threads to cores-- the workers, the Cilk workers-- to cores in any way that it feels like. This says, no, I want you to have it exactly here so that, when I run something else, it's exactly the same thing. And so forth. And we've already done a lot of this for you for AWS run.

By the way, there is no way of getting a completely deterministic result out of running on modern hardware. Does anybody know why? If I have a serial program and it's deterministic serial program, and I set it up and I reboot the system so it's got exactly the same content, et cetera, I still can get non-deterministic results. Does anybody know why? Yeah?

**AUDIENCE:** Because [INAUDIBLE]

**CHARLES E. LEISERSON:** No, you can turn that off. So they actually do randomize address space for security reasons. But when you run under the debugger and so forth, they tend to turn that off so that you can get repeatability for debugging purposes. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** No, those are generally-- no, those are deterministic, but there is something in the hardware which is non-deterministic. Does anybody know what is, happen to know, can guess what it is? Something the hardware that's non-deterministic. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Disk access would be, but if I'm just running something in core-- I'm not using disk, I'm just going to run-- it turns out non-deterministic, even though I'm just running everything inside with ordinary DRAM memory, and so forth. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** No, the out of order execution is all deterministic. There's no randomization there. Yeah?

**AUDIENCE:** Branch prediction.

**CHARLES E. LEISERSON:** Branch prediction's all deterministic algorithms-- all deterministic. Yeah?

**AUDIENCE:** The system clock signal?

**CHARLES E. LEISERSON:** System clock signal's very regular. Very regular, if you turn off DVFS and stuff like that. Yep?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** So the Linux schedule is, in fact, a deterministic algorithm. And if you're just running on one core, the scheduler never comes into it. So there is one source of non-determinism, and that is memory errors. So there's a chance that one of your memory-- when you're accessing the DRAM, that an alpha particle collided with one of the bits and flipped it. And there's hardware in there to do error correction, but it takes an extra cycle to do it.

So if it reads the memory location, discovers that there is an error in what it read, then it performs the correction, and then you get it. And that's something that's completely non-deterministic, because its alpha particle's coming from outer space-- space aliens messing with your system. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** No, actually-- now, most cache is covered by error correction, but the most likely-- if you look, the biggest memory is the DRAMs. That's the most likely source of these things. So in any case, I want to now just talk about a few things which-- just some examples of things that might come up for you. So these are ones that I've mentioned, but I just want to go through them in a little bit more depth.

So one of the things is code alignment can make a difference. So what happens is you have your program, and you make a change that you think is improving your program. But what happens, let's say, is that it causes an extra byte to be put into the code. So maybe the compiler is very smart. You made some little change. Then everything that follows it gets shifted down, and so the cache alignment issues can be completely different. Something can go across a page boundary that didn't used to go across the page boundary, and that can have a big impact on your performance.

This is like, yikes. This is like, yikes, how am I supposed to-- maybe we should just pack up on performance engineering. So everybody gets what the issue is there? So in this case, I inserted one byte. Well, everything after that, it's all linear in memory would change. Here's

another one that's even more insidious. If you change the order in which the .o files appear on the linker command line, that can actually have a bigger effect than going between minus 02 and minus 03.

And when you compile, you compile this order versus this order. You can have actually quite a big difference. Yike, OK, so what do you do? Well, first of all, one of the things I'm glad to say is that the compiler people have recognized this, and the situation is not as dire as it was years ago. What they do now very often is do a lot of alignment already. So for example, it's common for compilers to produce-- to start every function on the first word of a cache line.

That way, when things get slid down, you might apart from one cache line to the next for where it starts, but you're not going to affect the-- where you lie on the cache line, which can make a difference, by the way, in branch predictors, and things like that. And so that helps. That really quiets a lot of things. And in fact, they give you some directives. So LLVM has these switches. As far as I could tell, the first one, which is align-all-functions, I think-- I was unable to test this in advance, but I suspect that it's actually already doing this and this is actually a no op, because it's already aligning all functions.

I don't know that for a fact, but you can give the switch anyway, which will help if I'm lying. So that forces the alignment of all functions. So all functions start on the cache line, and that way, if you change one function, it's unlikely to-- it won't change the cache alignment of another function. It will only change the cache alignment of the function that you're messing with.

You can also ask it to align all blocks in the function. So remember that, in LLVM, we have these basic blocks, these pieces of serial code with links, the basic blocks. So what it will do is force every one of those to be on a boundary. But of course, what that means is now you've got a jump between these codes, even if it were going to be the next instruction or put in a bunch of no ops. So that can substantially increase the size of your binary, and it can slow you down. But on the other hand, you'll get very reliable results from then on, because every block is now cache-aligned.

Probably more practical is to align-- is to force the alignment of all blocks that have no fall-through predecessors. That is, you don't have to add notes. So this basically reduces it to the ones that are usually causing the trouble. So a line of code is more likely to avoid performance anomalies, but it can also sometimes be slower. And so one of the questions is, well, which matters to you in your particular thing?

Here's one that I love. So the example that I gave before of the order of linking, we have that as a reading assignment, by the way, on the web, that paper. And here's another result from this paper by Mytkowicz and his colleagues. They have a wonderful paper called *Producing Wrong Data Without Doing Anything Obviously Wrong*. So one of the examples they give [INAUDIBLE] the ordering of things, saying that, look, that actually had an impact between 02 and 03. The program's name can affect its speed, and here's why.

The executable's name ends up in an environment variable. So when you start up the program, it puts it in an environment variable. The environment variables, it turns out, end up on the call stack. So you can find out from the program what's the name of the program I was invoked with. The length of the name affects the stack alignment. You have a longer name, they put longer-- a longer piece there. And so if the data happens to go across page boundaries-- some critical piece of data goes on two pages, rather than one-- that could have a big impact on your performance. And there are other data alignment problems that can arise.

Yeah, this is kind of nasty stuff. So as I say, what we've done-- we've given you a very quiesced system. We have done everything we can to make it so that, when you submit stuff in this class, it's measured. But as we get along in the semester and you're doing more sophisticated stuff, some of this is going to fall increasingly to you to do things that make it robust.

OK, let's move on. We've talked about quiescing. Now, let's talk a little about tools for measuring software performance. So I did a little thinking, and I figured that there's actually about five different ways I know of measuring the performance of a program. So one is you can measure the program externally. You can run the time command. If you say time and then give a shell command, it will run the shell command and then tell you at the end how long it took.

You can instrument the program. That means to actually put timing calls into the program. So you can use things like `clock_gettime`, which I recommend. And we'll talk a little bit about these other two, `gettimeofday` and `rdtsc`. And you can either do this by hand where you say, I'm going to time something myself, or it turns out you can have compiler support for taking timing measurements. But what that involves is changing the program in order to put these timing calls in. And of course, you have the effect that, if you're perturbing the-- if you're putting these timing calls in, you can be changing the timing. So you've got that problem to worry about, as

well.

One of the ones is you can interrupt the program. One of the cheapest ways you can do it's just basically take gdb. So you start running your program. You run it under gdb, and then you type Control-C. And then you look. Where is the program? And you do that a few times, and you say, hey, the same routine is always where the code is running. Oh, that must be where it's spending all its time. That's actually been put into a thing called poor man's profiler, and then you can actually-- this is essentially the method that gprof uses. And we'll talk about that in a little bit.

You can exploit hardware and operating system support. So there are a bunch of hardware counters that the operating system and hardware support-- that, for example, perf uses. And so you folks are familiar with the perf tool set. Those are basically using hardware and operating system support. And you can also simulate the program. You can run it as a simulation, and then you really can go in and do anything you want to understand.

But of course, it's much, much slower, and you hope that your simulator is modeling everything that's have relevance to you. Maybe it's not modeling something that turns out to be relevant for you. So I'm going to talk it give some examples of each of these five things. So let's start with the time command. So it can measure elapsed time, user time, and system time for an entire program. Does anybody know what these three terms mean? Elapsed time, user time, and system time. Sure.

**AUDIENCE:** Is elapsed time [INAUDIBLE]

**CHARLES E.** Yeah, it's what we call wall clock time. Good.

**LEISERSON:**

**AUDIENCE:** User time is how long a program runs [INAUDIBLE]

**CHARLES E.** It's in the kernel working on your stuff, as opposed to somebody else's. So that's exactly right.

**LEISERSON:** So when you run the time command, you get some numbers like this where it says here's the real time, here's the user time, and here's the system time. Now, you might think that the user time and the system time should add up to your total time. Uh-uh-- doesn't work that way.

And part of the reason is that it may-- the processor may actually not be working on your code. You may be contact switched out and something else is in. So in any case, we have those

three types of things-- user clock time, the amount of processing time spent in the user mode code within your process, and the systems time, which is the time spent in the kernel, but within your process-- for example, satisfying system calls and such.

Now, the timing call that I recommend you use, and that was used in the example that I gave, is `clock_gettime`. In particular, there are a bunch of options to that, and the one that I strongly recommend is `CLOCK_MONOTONIC`. And it takes about 83 nanoseconds to actually read what the time is. That's about two orders of magnitude faster than a system call. And one of the things about it is-- this is such a funny thing to have to say-- is that guarantees never to run backwards.

Turns out the other timers can run backwards. You can take measurements and discover they're negative. This one does not run backwards. Part of it is because some of the other timers do things like, oh, there's this national standards thing that, periodically, your computer goes out to find out what the real time is, and it resets its clock to be consistent with whatever the global clock is. And that will cause the clock to be updated in a non-standard way, and where suddenly you lost some time or gained some time.

So this is really the [INAUDIBLE]. The only unfortunate thing about this is that it is non-deterministic how long it takes. And let me explain a little bit what's going on in this. So what happens is it takes a measurement-- it has to take two measurements to figure out what the elapsed time is to find out what the actual time is. It can't just take one measurement because it may have been swapped out. And the kernel helps support, in user space, something that says, here's the total amount of time you've spent up until you started your time slice.

So when you read that, you have to read those two values. Well, how do you know that you don't have an atomicity violation? You read one of the values, you got switched out, you get switched back in-- now, you have a new value for the other one. So the way it does it is it reads this register. It reads what the operating system has kept as its cumulative time, it reads the clock, and then it reads that register again. And if those two things differ, it knows there's been a switch in there. If they're the same, it knows there isn't, and that the number that it can take is reliable.

So in that kind of case, it will actually take two measurements-- more than one measurement. You do it again and you could have another context switch. And you could do it again, and have another-- but this thing is generally pretty fast. And on my laptop, it takes about 83

nanoseconds to run. There's a lot of people say, well, why don't I just read the cycle counter? That's actually cheaper. It runs in about 32 nanoseconds. And that you can do with the rdtsc-- read the timestamp counter-- instruction.

And you can do it yourself by using a built-in assembly. And what it does is it returns to how many clock cycles since the boot, and it runs in about 32 nanoseconds. But why not use this? Well, one thing is that rdtsc may give different answers on different cores on the same machine-- so the cycle counters on a processor by processor basis. Sometimes tsc runs backwards, as I mentioned. And also, the counter may not progress at a constant speed.

So remember that the time between-- the system is possibly slowing and speeding up the counters, and so forth. And converting clock cycles, for that reason, to seconds can be very tricky. So I recommend you stay away from this faster counter, this faster timer. The other one is don't use gettimeofday. That's the one most people know. That gives you microsecond precision. It's not actually microsecond accurate, but it gives you microsecond precision.

Because it has similar problems, whereas this particular-- the clock\_gettime MONOTONIC has been very well engineered, in my opinion, to give good reliable numbers at a reasonable cost. Any questions about that, about taking measurements and what to use? This stuff, by the way, over time, it's going to change. People are going to come up with better ways or worse ways. Or they'll say, we're not going to support that anymore, or what have you. And then, if you're out there as an engineer, you're going to be on your own. Hopefully you know what some of the issues here are and you're prepared to be on your own. Yeah?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** So when it does that, it aggregates and the operating system has to do it. Those numbers actually are very-- are relatively coarse grained. You cannot time something that's very short-- with time, for example-- with the time command. In general, my experience is you should, even with something like-- you can get very fairly fine grained measurements with the clock\_gettime. You can get fairly good measurements there, but unless you're aggregating, unless you're running code that's running around a second, certainly at least a 10th of a second-- if you're not running things that are that long, you run the risk that you've got really, really bad-- you get unlucky.

Let me point that out in this particular example. So here's the interrupting strategy, which we



talked about briefly. This is where I just Control-C at random intervals. And you look at the stack and say, who needs a fancy profiler? In fact, there are large companies who use this for debugging their big, big codes. Facebook comes to mind. They actually use this technique. It is a really easy worthwhile technique for figuring out where time is being spent.

Now, there are some other things. The Poor Man's Profiler-- and people have built things like gprof and so forth to increment-- to automate the strategy so you get this information. Because then it automatically [INAUDIBLE] looks at the stack, what's being executed, puts that into a call graph and so forth, and figures out where the time is going. But neither of those programs is accurate, if you don't obtain enough samples.

And just to give you an example, gprof samples only 100 times per second. So if you're going to use gprof for timing something that's only a second long, you only got 100 samples. How many samples is 100? That's not actually a real lot. And so you get wildly inaccurate numbers from interrupting. But on the other hand, for a quick type-- we use gprof all the time. It's quick. We do Control-C all the time. It's really quick. I don't have to install anything. I just take a look, and it gives me a first cut at what I want to do. It all depends. You don't need all the surgically precise tools all the time. Sometimes a really dumb tool is adequate for the job, and a lot quicker to deal with.

Hardware counters-- so one of the nice things that's happened in recent years is that there has become available a library called libpfm4, which is virtualizing all the hardware counters so that you have access to them with-- to all of these types of events on a per process basis. So normally, there's the hardware counters, but then you switch to some-- if there's context switching going on to some other process, then what happens to your counters? They have to be saved, they have to be updated.

So anyway, libpfm4 does all of that kind of virtualization to make it so that the counter-- you can view it as if it's your own counter. And perf stat, for example, employs that. There are a lot of esoteric hardware counters. And as I say, good luck in figuring out what they all measure, because they often are not well-documented. A few of the important ones are well-documented, but most of them are very poorly documented exactly what it does.

A really good example was we had somebody who was looking at cache misses to figure out how much bandwidth-- so last level cache, L3 cache misses to count how much data was being transferred from DRAM. And they were getting curious numbers that didn't seem to

measure up. And it's like, wait a minute, have a miss, moves from DRAM onto the chip. Why is that not counting how much stuff is being moved, if I count up how many cache misses times how many bytes in the cache line? Which is what, on the machines we're using?

How many bytes in a cache line on the machines we're using? OK, 64. OK, gotcha, you guys are-- OK, 64. But not every machine has that. But anyway, so why was this not measuring how much stuff is being moved? I'll give you a hint. It used to measure how much stuff was being moved, but then those architects, they are such pesky, clever people, and they put in a great feature.

**AUDIENCE:** Prefetching.

**CHARLES E. LEISERSON:** Prefetching. They put it in prefetching. There's things that fetch it, and it doesn't update that counter. So if you want, you have to count how many prefetching incidents you have, as well. So you can often cobble this together, but good luck figuring out what some of these do. Also, watch out. You may think that the tools let you measure a lot of counters, if you want.

But if you read the fine print, it turns out that, if you do more than four or five, it starts essentially time sharing the available counting bandwidth that it has, and it's not-- it's actually just doing something statistical, rather than actually counting them. So you can't count more than like four or five-- actually, four or five I think is a high number. But somebody I know well, who knows this stuff, said four or five is probably what it is today. So that's hardware counters. So hardware counters are good technique.

Next one is simulators. So things like cachegrind usually run much slower than real time, but what's great about simulators is you can get repeatable numbers out of them. You run the code. You run it again. If you've set up everything right, you can get-- and you can see what's going on inside. The downside is that they don't necessarily-- it's slower, and it doesn't necessarily model everything going on in the cache.

But for things like cache misses, this is a great tool to just figure out what's the fundamental cache-- and we'll talk about that when we talk about caches in the next couple of weeks. And if you want a particular statistic, in principle, you can go in, and if it's an open-source simulator like cachegrind is, you can collect it without perturbing the simulation. So any question about these ways of collecting measurements?

There are a whole bunch of ways of doing it-- they all have pros and cons. They all can be

useful in a given context. They all have some flaws. A really good strategy-- I'll talk about this later-- is triangulation. I never take one measurement and believe it. I always want to take at least two measurements in different ways, and make sure they're telling me the same story-- triangulation. If there's a discrepancy, then I want to know what's causing the discrepancy. But I never trust one number, and I never trust any numbers without having a model for what I think is coming up.

And in fact, that's what we're going to talk about next is performance modeling. So any questions about measurements and such? Isn't it good we have AWS run? Number comes back, it's the number. And it's actually a pretty good number. We've worked very hard to make that a pretty good number. So performance modeling-- so yeah, we cover a lot of stuff in this class, as I think some of you have started to notice.

But really, performance-- software performance engineering is pretty simple process. You take a program that you want to make go fast, program A. You make a change to program A to produce a hopefully faster program A prime. You measure the performance of program A prime. If A prime beats A, then you said A equals A prime. And if A is still not fast enough, you repeat the process. That's basically what you're doing. It's pretty simple.

And as should be apparent, if you can't measure performance reliably, it's hard to make many small changes that add up, because it's hard to tell whether A beats A prime-- sorry, whether A prime beats A. It's hard to tell. And so as a consequence, what we want to do is we want a model of performance that we're in a position to draw accurate conclusions. So we want to do things like drive the variability of measurement down to 0. And we want to do things like figure out ways of using statistics to give us more accurate picture of what's going on than maybe what is apparent. And that's basically what we're going to talk about for a little bit.

Part of what I'm going to do is going to talk about statistics. How many people have had a statistics class, or a machine learning class, or something where you dealt with statistics? Beyond probability, I mean. I mean real statistics, sample means, and things like that. So a few of you. OK. The basics of what you need to know you can find from Wikipedia, or MathWorld, or someplace like that. And I'm not going to try to teach you Statistics 101, but I will point you in some directions, and give you some pointers to some tools you can do.

OK, so here's a puzzle. Suppose you measure the performance of a deterministic program 100 times on a computer with some interfering background noise. What statistic best

represents the raw performance of the software? Is it the mean of the-- the arithmetic mean of those runs? Is it the geometric mean of those runs? Is it the median of the runs? Is it the maximum of the runs? Is that the minimum of the runs? This is where it's helpful, if we had those clickers or whatever they have. But we don't, so I ask people to vote.

Now, I want everybody to vote. And once again, it doesn't matter if you're right or wrong. You can be right for the wrong reasons. You can be wrong, but have the idea right. But it's fun when everybody participates-- certainly more fun for me when I see hands go up than when I see people sitting there looking bored. OK, how many people think arithmetic mean would be a good way of measuring the raw performance? Arithmetic mean. That's the most common statistic that we ever gather.

OK, what about geometric mean? OK. What about median? OK, good. What about maximum? One for maximum. What about minimum? OK. So turns out that actually, these are all good measures to take, and it depends upon what you're doing with them. But turns out minimum is-- does the best it noise rejection. And that's because you expect any measurements higher than the minimum, if it's a deterministic program and so forth, that's going to be due to noise.

So if you're really interested in knowing how long fundamentally your code takes on the underlying hardware, when there's other things going on, taking the minimum rejects it. Now, you might say, the median also rejects noise, but it doesn't, because if you view your program as being its running time plus noise, then the median is going to give you some number that is in the midst of the noise. It's going to have some component. So minimum is the only one that really rejects all of them.

But they're all useful measures in different contexts. And there are ways that you can use the mean and some of these other ones, as we'll talk about in a minute, to get information about making decisions, because is the thing that we're after is not necessarily always the raw performance of the software. Sometimes we're interested in whether A beats B. That's a different question than how fast is this fundamentally go.

Now, so there are a lot of different types of summary statistics, and there's lots of reasons to pick different ones. So for example, if you're interested in serving as many requests as possible in a web server, for example, you're going to be looking at something like the CPU utilization and taking the arithmetic mean to try to understand how those things add up. If all tasks have to be completed within 10 milliseconds, then you're going to look at the-- you're

looking at the total, and you're going to add it up, and you're going to be interested in making sure that each one is small. And that's also what the mean does. And you're going to be looking at wall clock time.

If you want to ensure that most requests are satisfied within 100 milliseconds, you might be looking at the 90th percentile behavior. And you'll say, yes, I won't make every one, but I want 90% of the time I want to get it there, and I'll be using something like wall clock time. In a lot of web companies, there's a thing called a service level agreement. This is what they should give you for your telephone, but they don't. Tells you what kind of service you can expect, and if they don't meet that service requirement, then they haven't lived up to it.

Instead, we buy these phones and we get the service, and we just get whatever they decide to give us. But if you're a big company, you insist that you get some kind of service out of the people that you're using. And so there that's typically some weighted combination, and you're using multiple things. You might want to fit into a machine with 100 megabytes of memory, some sort of embedded machine or whatever, then you're going to be interested in maximum of the memory use. So it's not all the performances, not all just time.

You might want the least cost possible, and you're looking at things like energy use, et cetera, or the fastest, biggest, best solutions. You can see average comes up a lot as one of the ways. So I wanted to though cover one particular example, which I find is the most commonplace I see a misuse of summary statistics, and that's for when I'm summarizing ratios. So suppose I have two programs A and B, and I run the four trials. Normally, you'd run a lot more, but I wanted to fit on a slide.

And program A, on whatever trial 1 was, took nine seconds, say. On trial 2, it took 8-- 2 and 10. And program B you've got 3, 2, 20, and 2. So I can compute the mean for each of those. So the mean of the program A is 8.25, and the mean of program B is 4.75 over those four benchmarks. I can also take a look at how much is A winning-- sorry, is B winning over A. And so if I take the ratios, I then get 3, 4, 1/10, and 5 for a mean of 3.13. It's actually 3.125, but I'm only keeping things to two digits.

And so if I was asked to summarize this, I could perhaps conclude that program B is more than three times better than program A, based on these statistics. But there's a bug in that reasoning. Can anybody see the bug? This is wrong.

**AUDIENCE:** [INAUDIBLE]

**CHARLES E.  
LEISERSON:**

It doesn't make sense to take the arithmetic mean of a bunch of ratios. Why's that? Yeah, one thing here is that you can see is that-- notice that the mean of the ratio is not the same as the ratio of the mean. That should be suspicious. Should I not be comparing the ratio of the mean, or should I be comparing the mean of the ratios? So that's not particularly good.

Another thing is suppose I take a look at the ratio B over A, and I take the arithmetic mean. Then what I discover is that A is better by a factor of almost three than B. So clearly, taking the average of the means is the average of the ratios. There's something wrong with that. And in particular, as I say, the ratio of the means is not the mean of the ratios. And then your intuition is spot on.

Suppose instead I compute the geometric mean. So geometric mean is basically like taking the average of the logs, the arithmetic mean of the logs. So you're basically taking the product and taking the n-th root of the product. And I've computed that for these things. And now I've taken the arithmetic mean of the run times, because that makes sense. That's kind of an average over these things, how long did things take. And now I look at A over B and B over A, I get the same thing. And it's, in fact, the case that the ratio of the means is the mean of the ratios.

So when you look at ratios, there's another place that comes up where you look at rates. And I've seen people look at rates, and it turns out when you're looking at rates, often it's the harmonic mean that you want in order to preserve these good mathematical properties, and make sure that-- it's really bad if I do this thing and I say, look how much better A is than B, and somebody-- if I flipped the ratio, B would be better than A. That's really suspect

If the identity of whether you're A or B, and which one was the numerator, which in the denominator had an impact, that would be really suspect. So this is the kind of thing, when you're thinking about how you're reporting things and so forth, you want to be careful about, when you're aggregating things, to make sure that you have the basic mathematical properties met. And what's nice is 1 divided by 1.57 here is, in fact, 0.64. So it didn't matter which way I took the ratio-- I got the same answer. So B is better that by a factor of about 1.6, something like that. It was a little bit better. It's not three times better. And it for sure isn't the case that A is better than B by a factor of three. OK, good.

Any questions about that? Good, OK. Suppose as I want to compare two programs, A and B,

which is faster, and I have a slightly noisy computer on which to measure their performance. What's a good strategy? What's a good strategy for comparing and figuring out whether A is better than B, or B better than A? Sure.

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Sorry, so you're saying I'm going to run multiple runs? That's a great idea. We're going to do multiple runs. And what am I doing for each of these runs?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** The low order statistic of the-- for example, minimum or 10% or something really low. OK. So I take that. I have one number for program A. I have one number for program B. I ran them n times. What else?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Well, am I comparing the two minimums I've got there, or am I comparing each one?

**AUDIENCE:** [INAUDIBLE]

**CHARLES E. LEISERSON:** Oh, I see what you're saying. Take all the measurements that go below your 10%-- cheapest ones-- and then compare those. OK. So you're substantially doing noise reduction is what you're saying, and then other than that, you're comparing by comparing means. OK, that's an interesting methodology. You'd probably get something reasonable, I would think, but you couldn't prove anything about it. But that's an interesting idea. Yeah, that's an interesting idea. What else? Yeah?

**AUDIENCE:** You run both, and then see which one is faster and mark it. And then run them both again, see which one is faster, mark it again-- mark that one. And keep doing that, and then see how many marks [INAUDIBLE]

**CHARLES E. LEISERSON:** Good, so you're saying to a bunch of head-to-head runs, and mark just who wins over those things. So one wins more than the other, or the other wins more than the one?

**AUDIENCE:** Yeah.

**CHARLES E.** What good is that?

**LEISERSON:**

**AUDIENCE:** If the time it takes for each program to finish is a random variable that tells you [INAUDIBLE] or how much more [INAUDIBLE]

**CHARLES E.** Yeah. So this is actually a very good strategy and actually has some statistical muscle behind it. So what you can do is do n head-to-head comparisons between A and B. So in both these examples, yeah, we better run it a few times. And suppose that A wins more frequently. So now, what we want to do in the statistics, where they call the null hypothesis, is that B beats A. So even though we see A beats B, the null hypothesis that we're wrong, and that A beats-- that B beats A.

And what we then cannot calculate is what they call the p-value, which is the probability we'd observe that B-- that A beats B more often than we did. So for example, for each of these-- to imagine that in the worst case, let's just say that they were equal in performance, and all we're seeing is the noise. Then I would expect that I would get about an even number of things. And so the further that I deviate from them being even, especially once I-- that distribution is essentially a Bernoulli distribution. The further I get out-- or a T distribution, if you have small numbers.

As I get further out from-- away from the mean by what the variance of just flipping coins, in that case, I can calculate what's the probability that I'm seeing something that would be that extreme. And that gives me a good reason to reject the null hypothesis, if it turns out that deviates by a lot. So this is very standard stuff in the social sciences. Who's had a course in this kind of stuff, in testing null hypothesis, biology, and so forth? You remember anything from it? Just vaguely?

Yeah, yeah. OK, that's fine. One of the things I have found in life is that I don't remember anything I learned in college to speak of. What I had to do is relearn it, but I relearned it a lot faster than when I was in college. And so that's part of this is also-- so when you see it again-- oh, OK, now, you know how to go about learning this.

Once again, you can look this up on Wikipedia or whatever, if you want to use this methodology. It's a good one for being able to determine, even in the presence of noise. What it does do is it doesn't tell you what the raw performance was, but it says, in that environment, which one is actually faster. And that may actually be a more relevant question, because



you're not always going to have this completely quiet system. You may have a lot of noise going on in a system. You'd like to know which one is going to behave better in what that actual system is. And so this methodology is pretty good. Note that, with a lot of noise, we need lots of trials.

The last thing that I want to talk about-- which I won't-- is fitting to a model. And this is the issue of sometimes you measure things, but you're interested in a derived statistic, such as, in this case, gathering some time, counting instructions, counting cache misses, counting time, and asking, OK, what's the-- what can I estimate is the instruction time, and what is the cache time? And to do that, you do a least-squares approximation. And there's, once again, some statistics behind that. I'm not going to go over it here, but you can look in the notes.

There are a bunch of issues with modeling, which is that you can over fit very easily. If you add more basis functions, you will fit the data better. And so how do you know if you're overfitting? Answer is, if you remove a basis function, it doesn't affect the quality very much. Is the model predictive? I'm actually going to just let you guys look at these, because it's pretty good. I think they're pretty self-explanatory. Let me just finish with a couple of words from a giant of science.

This is Lord Kelvin. What is Kelvin famous for? Besides Kelvin. He was the guru of measurement. And he said, to measure is to know. That's a good one. And he said also, if you cannot measure it, you cannot improve it. So both very apt sayings from the same guy, so there's a reason he's got a big forehead, I guess. So anyway, that's it. Thanks very much. And good luck on the quiz on Tuesday.