GUEST SPEAKER 1: All right. So we're going to talk about performance engineering and pretty much how to do that with profiling tools. So first, we're going to talk about what profiling tools are and how you use them in general, what the theory of that is. And then, more importantly, we're going to do an interactive walk-through of two examples that we picked out. The first one is the Matrix Multiply that you guys saw in Project 0, the cache ratio measurements and how to do those measurements and how to do the cycles per instruction measurements, these kinds of things. And then we're going to look at doing some of the bit hacks that Charles talked about in the second lecture. We're going to work on a branchless sorting algorithm, which is pretty interesting.

So the code we gave you in project one, it was very obvious where the hot spot was. But real life doesn't really work that way. And code gets really, really big. It's not always obvious where the part is that you want to optimize. And you really don't want to waste time taking what is otherwise understandable, good code, and beating it over the head trying to make it go faster. So that's why Donald Knuth said, "premature optimization is the root of all evil." You really want to focus on the hot spots.

So that's what profiling's for. It basically gives you visibility into the execution of your code, because it would be too complicated to just sit down and look at it and try to understand where it's spending its time. And there are many possibilities for where it could be spending its time.

In 6-172 we're mostly going to be focusing on things that are CPU-bound and things that are using the memory hierarchy ineffectively. But in the real world, a couple of other common things are network requests. So if you've got your web application,

it's making a whole bunch of AJAX requests. It's going to be wasting a lot of your users' time. If you've got your desktop application, it's hitting the disk all the time. That's also going to slow you down. I think SQL database, if anybody's ever done web development, they know that can be a huge bottleneck.

So basically, the tool that you're going to use is going to depend a lot on what the problem you're solving is. But the basic ideas are all the same. There are all kinds of profiling tools for measuring these things and figuring out where it's coming from and where the time's going.

So for trying to figure out CPU and memory, there's a couple things you can do. The first and most basic and sort of most obvious to you guys is you can do manual instrumentation. And what that is is basically where you go in and you put in prints and try to figure out where it's spending its time, what codes could you have executed more.

Or you can be a little bit more clever. You can try to insert logging calls that fill up a buffer with checking the time and actually doing that yourself. And one of the big advantages of that is that, when you actually go into the code and you instrument yourself, you can cut down the information that's coming back out at you. If you have an intuition for how the program works, sprinkling little calls to timing functions around your code in the places that you think matter is going to give you maybe a better picture. But it requires more work on your part. So that's the drawback.

The other kind of instrumentation is static instrumentation, which is basically-- and I should say that instrumentation is basically a little snippet of code that is gathering information, gathering data about the code execution. So static instrumentation is basically code that's inserted by the compiler when you build your code. And gprof is an example of this. Basically, what it does is it will, at the prologue and the epilogue for every function, insert a little piece of code that increments a counter every time the function gets executed. I think it also figures out who called it, that kind of thing.

One of the drawbacks to static instrumentation is that you have to have the source,

and you have to be able to rebuild your project. If you've ever had a DLL binary blob of data library handed to you, and you want to know where the time's going in that library, you're not going to be able to instrument it. So you're not going to get any data.

So that brings us to the next thing, which is dynamic instrumentation, which solves that problem. Basically, what happens there is you take the binary as it is just before you're about to run it. And before you start executing, you actually just take a look at the code and translate it. And you go in, and you insert the instrumentation. And then you have this new snippet of code that you then run instead. And so that works for binary blobs. And it doesn't require a recompile, which is great.

But it introduces a lot of overhead, because you're not actually running the original code, right? You're running this translated thing, which might be much more inefficient. And I guess Valgrind has a couple of tools that are sort of classic examples of that, Callgrind and Cachegrind. I think Callgrind gives you a call graph. And Cachegrind tries to measure the cache effects, tries to make a prediction of when you do memory accesses, are these things in cache or not.

Another thing you might want to look at, or the thing that we're going to focus on in this course are performance counters. And two tools to get these are OProfile and perf. And basically, the CPU has a bunch of counters-- we've talked about them-- for branch misses, cache misses, these kinds of things. And this will give access to those.

The last tool that I guess I'm going to talk about for profiling is heap profiling. A lot of times, if memory's your problem, you really want to figure out who's doing all the allocation. And so these tools will give you a nice breakdown of who's allocating what, who's calling them, and who can I stop calling so that I'm allocating less memory. And like I said, you can find more tools for profiling different kinds of problems and resources.

But today we're mostly going to focus on perf, which is a new tool for getting access to performance counters on Linux. I guess I should also mention that, as someone

talked about in the last lecture, the way that you want to gather data from performance counters is you want to do event sampling. There's too much data. As you execute along, you don't want to record, OK, I executed this instruction, then I executed that instruction, and I'm going to count all the instruction executions and then store that data somewhere. That would be way too much data.

So what you want to do is you want to sample every so often, every 1,000th event, or some other frequency for whatever your interesting event is. And basically, what happens is that when that counter has hit the threshold, an interrupt will fire, and the kernel will catch it. And then it will record the data that you want, the context, like where in the execution it was, what the call stack looked like, these kinds of things.

I think we've covered most of the performance counters that we mostly care about. But I think it's worth mentioning that there are a lot of performance counters. And you can go look them up in this part of the manual. They cover a lot of interesting things, not just the ones we've talked about, including sleep states, power consumption.

And if you want to look at contention, say you have a parallel program, and you have two cores accessing the same data. If they're writing back to it, they're going to be fighting over the same cache lines. And so there are counters for things like, how many times did this core have to go off core to get cache line, and all kinds of interesting things like that. But mostly, we're going to focus on the cache misses, branch misses.

I also wanted to mention that the tool we're using, perf, it kind of replaces OProfile and PerfMon, which are tools that do similar things. I guess it's different in that it's a unified interface to monitoring two kinds of events. The first are software kernel events, so things like contact switches. So how many times did your application make a system call and then get contact switched out? And how many page faults did your application generate? These kinds of things that the kernel knows about, but aren't necessarily hardware events.

And the other kind is hardware events. And those are sort of the ones we're going

to focus on. We've already talked about those. If you want to get this tool and try it out yourself, if you have any recent Linux distribution, anything with a kernel that's less than a year old, you can install Linux tools. And the command is perf. So now we're going to do a demo of the matrix multiply from Project 0.

GUEST SPEAKER 2: All right. So for the demos, I guess I'll narrate what the code's doing, so Reid can focus on typing. This code should look familiar. All right. Can everybody hear me? All right, cool. So this code should look very familiar. It's Project 0's matrix multiply code, the original version. And it's a triply-nested four loop for multiplying matrices. So what we're going to do is I'm going to switch to a terminal that Reid is typing on. And we're going to run matrix multiply. And it takes about eight seconds to run.

And now we're going to run it, but have perf output some performance statistics for it, basically, CPU counters. So you use -e to request specific counters. You can use perf list to take a look at all the counters that perf supports. In this case, we're asking for the number of CPU cycles, the number of instructions, the number of L1 cache loads, and then the number of L1 cache misses. Perf stat. That's better.

So it takes a little bit longer to execute, I think, with perf. But it's not supposed to be a big difference. You can see, 8.31 versus 8.34. So it's relatively low overhead sampling, which is good, because you don't want to change the behavior of your program when you sample it.

Important things to see. So we're looking at this cache miss rate, which is 18%, almost 19%, L1 cache miss rate, which is pretty bad. So as a result of that, if you look at the number of instructions, the number of cycles, perf calculates the IPC for you. We previously talked about CPI. But perf uses IPC, which is instructions per cycle. So basically, every cycle, you only executed half an instruction on average, because you're waiting for the cache. So that's no good. So can anybody remember from Project 0 what you did to work around that? Yes.

AUDIENCE: You reordered the loops.

GUEST SPEAKER Yes. So the before and after. You just swap out the inner two loops. And we will run

2: it again. OK, so remember, the previous runtime was 8 seconds. Now it's down to 2.5 seconds, which is a lot better. Let's run the perf stat command again and look at the performance counters. And as you can see, the IPC jumped from 0.49 to 1.4. So it tripled. And if you look at the new L1 cache miss rate, it's 1% rather than 20%, which is a lot better. Yes?

AUDIENCE: Can you point to where the cache miss rate percent is? Because I just see zero per second.

GUEST SPEAKER 2: Oh, sorry about that. So L1 D-cache loads is all of the times that the CPU asked the L1 cache for something. And then L1 D-cache load misses is the number of cache misses. So you divide this over this to get the ratio, which is basically what Reid did over here. Does that make sense?

AUDIENCE: Yes.

GUEST SPEAKER 2: OK. So this is a ridiculously simple example of how to use perf, just so that you can see the command set. But that's not really a very interesting example. So the next one that we'll look at is actually something that Charles, Reid, and I did yesterday afternoon for fun. It started, we were testing your Project 2.1, one which is-- I don't know how many people have looked at the handout yet. But it deals with a bunch of mystery sorting algorithms that you have to figure out what they're doing.

And so we took one of them. And we were running sorting. And Charles asked, what happens if you have a branchless sorting algorithm? And we said, we don't know. So we played around with it. And in the process, we optimized our branchless sorting algorithm by using perf to generate insight into what the CPU is doing. And as you'll see, we generated quite a good speed-up.

So the first thing we'll do is, let's start with the baseline, quicksort, everybody's favorite sorting algorithm. So I'll switch over to the terminal. And Reid is going to run quicksort on 30 million integers one time, which it says takes 4.14 seconds. Now, let's use perf stat to see what it's doing. So this time, we will ask for slightly different events. We'll still ask for the cycles and instructions to get our IPC. But we'll also ask

for branches and branch misses, because we kind of intuitively know that quicksort is doing branching.

So OK. It's doing 0.8 IPC, so 0.8 instructions per cycle. And the branch miss rate, so branch misses over total number branches, is 11.5%. That's pretty bad in terms of branch missing.

So let's take a look at the code and see why that's the case. So here's the part of the code in quicksort where you select a pivot recursively, and then you loop, and then you do your swapping. So these branches are more or less unpredictable. And looking at them, there's no way that we could think of to get rid of the unpredictable branches. They're unpredictable by nature. So this would not be an interesting example of which to do branchless sorting.

So let's switch to a different algorithm, mergesort. So here's the relevant code in mergesort that takes a list, C, and two input lists, A and B. And you merge the contents of A and B into C by taking the smallest element from either A or B, putting it in C, and then repeating that process until the lists are empty. So the "if else" branches, as the comment says, tries to place the min of the value either at A or B into C, and then properly increment the pointer and do those housekeeping tasks so you don't crash.

So we're going to try running this and profiling it. So Reid is running mergesort. Took 5.04 seconds. Well, the last one took 4.8 seconds. So already, it doesn't look too good. But let's try to see what happened. So we'll issue the same perf stat command, asking for branches, branch misses, cycles. Reid is being lazy.

OK, so the IPC is 1.1, which is a little bit better. But if you look at the branch misses over branches, they're off by roughly an order of magnitude just like before, so 10%. So OK. Mergesort is currently slower than quicksort. And the reason is still, roughly, you still have all of these branches that you're not predicting correctly.

OK, so why branching, and why not caching? Well, we did some quick back-of-the-envelope calculations, or research, rather. And it turns out that in the Nehalem, the

processes that you're using on the clouds, one of the design changes that Intel made is that they made the L1 cache and, actually, also the L2 cache faster, so fewer number of clock cycles to go out to the cache. But at the same time, they also made the pipeline deeper.

So an L1 cache miss is maybe three or four cycles. An L2 cache miss is 15 cycles. By the time you miss the L2 cache, you're going out into L3 cache, which is 12 megabytes. And if you're sorting integers, most of them are probably going to be in there anyway.

AUDIENCE: John, those are actually hits [INAUDIBLE].

GUEST SPEAKER 2: Oh, they are. OK, so misses might be a little bit bigger than that. But OK.

AUDIENCE: L1 miss is a L2--

GUEST SPEAKER 2: Right, is an L2.

AUDIENCE: L1 misses.

AUDIENCE: It was just a typo.

GUEST SPEAKER 2: OK, so the next number is probably, like, 30 to 50, somewhere on that order. 50-something. OK. But a branch mispredict will also cost you somewhere between 16 and 24 cycles, depending on how many pipeline stages are actually in the Nehalem, which we don't know. So bad branch predictions might just be as costly as bad memory access patterns. And plus, Charles is going to go into detail about memory access patterns and how to optimize sorting for good memory access. So it wouldn't be interesting if I talked about that right now.

So let's optimize branching. So to do that, we'll use a bit hack that Charles introduced in an earlier lecture. So remember, all we want to do is we want to put the minimum of either A or B into C. So right now, we're using branches to do it. But there's no reason why we have to. So the trick that will apply is the XOR trick for

calculating min without a branch. Does everybody kind of follow what that code is doing? You can refer to the bit hacks lecture later to check. But trust us that it's correct.

GUEST SPEAKER 1: [INAUDIBLE]

GUEST SPEAKER 2: All right. I guess we can go over it. Do I have a mouse pointer? Cool. OK, so inside, you're doing a comparing A is less than B, the value of A is less than the value of B. And you store either a 0 or 1 into a flag called CMP. And then you take CMP, and you negate it. So then you end up with a bit mask of either all 0's or all 1's, depending on which one was smaller. And then you mask A XOR B with that. So this inner expression, you either get A XOR B out of it, or you get all 0's. And then you XOR B into it.

So in one case, B XOR this expression cancels out the B. You'll just get A out. And the other way, B XOR 0 is going to give you B. So that should be convincing motivation that this is actually a min function. And then, once you have that, you can do cute tricks to recycle the flags, to do the A++ and B++ and so on, using CMP.

So next, let's run this code and see what happens. OK, so we went from 5.04 seconds to 4.59 seconds. And we'll do a profiling run just to get some stats on it. OK, so now look at the IPC. We went from 1.1 to 1.7-- 1.7. And branch misses went down by a factor of 10, which is really awesome. But overall, the performance didn't seem to improve by much.

Well, if you want to look at why, look at the total number of instructions executed. That's that number versus that number. We seem to be executing considerably more instructions. So to see why that's the case, we decided to use another mode of perf called report, or record and then report. So what this does is that it logs where all of those interrupts actually happen. So it logs which point in your code the CPU was executing when these performance counters tripped.

And then it generates a nice report that we can call using perf annotate. And the top

section gives you a sorted summary, where it shows you the percent of times that perf caught your program executing that line of code. So 11% of the times that perf triggered its interrupt, your code was executing mergesort.c line 32.

So let's go to mergesort.c line 32 and see what it's doing. So this annotated view shows you the lines of C-code, which are in black. C-code is in black. And then line numbers are annotated here. And then it actually goes ahead and shows you the assembly instructions that it's executing. And it puts the timing actually on the assembly instructions corresponding to every C expression. Yes?

AUDIENCE: Why is this [INAUDIBLE] is the annotated version--

GUEST SPEAKER 2: Yeah, you call perf annotate and then a function name. The Project 2.1 will walk you through doing this a couple of times. So you'll be able to see what it's doing.

OK, so the assembly here is actually pretty important. I guess everybody's taken 6004, or some variant, where you had to code some form of assembly? OK, it's a prereq. Cool. So Saman will talk more about assembly. But in this case-- or Charles, actually, now will talk more about assembly. But anyway, in this case, it was actually kind of useful to look at the assembly. We won't expect you to know how to write assembly for this class or expect you to write high-performing assembly. But sometimes taking a look at what the compiler outputs can really help you.

Well, in this case, what caught our eye was this cltq thing, because none of us in the room knew what the heck cltq was It wasn't an assembly instruction that you usually see. So that caught our eye. And we wondered what was this assembly doing. So we spent a little bit of time tracing through this, working out what the compiler was doing.

So in PowerPoint, I pulled out this code. And OK, so I looked up cltq in my 600-page Intel manual. Yes, your processor does come with a manual. So cltq, it says, sign extends the EAX register to 64 bits and then places that into RAX.

Why is it sign extending? OK, so let's try to trace through what the assembly's doing. So here, we wanted to do comp equals star A less than star B. So it moves

the value at the address in register 14 into RCX and does the same thing for register 13. So assume that R14 and R13 are A and B. And so RCX and RDX are star A and star B, right?

And then it XORs ESI with ESI, which is 0. So it zeroes out the register ESI. And then it does a compare between RCX and RDX, which is the value at A and the value at B. So it does that comparison. And then, depending on the result of that comparison, it writes that result out into register SIL--

GUEST SPEAKER 1: Which is the low-byte--

GUEST SPEAKER 2: SIL, as Reid said, is the low-byte of the register ESI, which is used here. But before that, the compiler then executes this expression here, B XOR a, which it puts into register RDX. And then, finally, it takes ESI, which contains the same value as SIL, and then it moves that into EAX. And then it negates EAX. OK, so that kind of looks like negative CMP, right? And then, after negating it, it then sign extends it to a 64-bit register. And then it does the en masse that we asked it to.

Well, why is it jumping through so many hoops to do this? Well, it turns out, when you declare a value of type int, you say that it's a 32-bit value. And the compiler took you very literally in saying that you requested a 32-bit value. So it did a 32-bit negation. And then it extended that to 64-bits. So it wasted a couple of instructions doing this, instead of just flat-out doing a 64-bit negation to begin with.

Now, the problem with this is usually you don't care if it generates one XOR assembly instruction. But this is a tight loop that's called in the recursion process for every single merge that it's doing. So the time that it wastes here actually really adds up.

So how do we fix that? Well, instead of declaring it as int, let's just try replacing it with long, and then recompiler the code and see if things are any different. So here we start our process of trial and error. So we compiled mergesort long. And the runtime is 4.05 seconds. Before, it was, like, 4.5 or so. So the runtime definitely

improved. Let's compare it.

GUEST SPEAKER 1: Yeah, just to verify.

GUEST SPEAKER 2: So it went from 4.59 to 4.05, which definitely is an improvement. So let's figure out, well, let's see if it generated better assembly. So we'll run perf record in order to look at perf annotate again. And OK, once again, it's spending 14% of the time somewhere in mergesort.c, which is hardly surprising. We'll go to line 27 and, yeah, use long. OK, so that's roughly the assembly.

And Reid, although he wasted time to find it, I have a nicely prepared screen shot of the same section of code. So it looks cleaner. It loads A and B into different registers this time, actually, RSI and RCX. And then it clears out EDX. And then it does the compare between RSI and RCX, which is A and B. And then it sets percent DL. Now, percent DL is the lower byte of EDX. So it used a different choice of registers. And then it did the A XOR B. And then it moved that compare into RAX.

GUEST SPEAKER 1: [INAUDIBLE]

GUEST SPEAKER 2: OK, and Reid actually has a longer snippet of the code. And the compiler, actually, now that we told it that it was a 64-bit flag, the compiler realized that there's a couple optimizations that it could do. So it switched around the order of things a little bit. And it's recycling the value of RDX for loading, I think it's A+ equals comp. It's recycling the value RDX that it generated out of a compare.

And now, down there, it's negating RAX as a 64-bit number and directly using it without that cltq instruction. So that was kind of nice. So in just nudging the compiler a little bit, we got it to produce code that ran roughly about 10% faster, which was surprising to us, actually, that changing a single int to a long could reduce the runtime by that amount.

So next thing that we did was we scrolled down a little bit more on this window. And we looked at the next thing that the compiler was doing. And we kind of caught the

compiler again doing something silly. So here's the code that we were at before, the compare code. And now let's look at how the compiler did B+ equals CMP.

Once again, the reason that we stopped and looked at this code was we saw an instruction that we didn't recognize, SBB. Well, we looked at the manual again. And SBB of R1 and R2 is a subtract with a borrow bit. So it does subtracting the two arguments and then subtracting a carry flag from that.

So OK, why's it doing that? Well, let's walk through what the compiler tried to do. Let's see, where's a good place to start? So it's RDX in this compare. So it compares RDX to 1. RDX is where CMP was stored. So it checks whether or not it equals to 1. And then the result of that check is actually stored in the carry flag, CF.

And subtract with borrow, RAX, RAX. Before, RAX held the value of B. But now, since it did this, RAX minus RAX is 0, right? And then minus the carry flag is either 0 or negative 1, depending on the value of the compare. So once again, it generates a register that's either all 0's or all 1's, 64 bits.

Now, what did it go and do with that? Well, jump down a little bit later on in the code. And it ends EAX, which is the 32-bit version of RAX, with the value of 8 in hexadecimal. So now it's either all 0's, or it's 8. And then it adds that value, EAX, which is the same as RAX-- RAX is a 64-bit version-- it adds that value to RBP, which stored the address of B.

So it jumped through that many hoops just to increment B by either 8 or 0. Now, this seems a little bit unnecessary. So we thought, how else could we express B+ equals not CMP, and try to change the code around a little bit so that it does the same thing, but hope that the compiler treats it differently. Well, in this case, not CMP, when CMP is either 1 or 0, is the same thing as 1 minus CMP. Well, we compiled a version of this code, and then we tried to run it.

Refreshing our memory, mergesort minus, where we just did the minus sign. So we took the runtime down from 4.04 seconds to 3.77 seconds. And once again, this is just from replacing a not CMP to a 1 minus CMP. That's all the code change that we

made. And we will look at the annotation again, just to make sure that the compiler generated different code. Rather, we were curious of what the compiler generated this time. So it's roughly this section of code.

Now, one thing you'll notice is, because we're compiling with-- this is already compiling with GCC's maximum optimization level, mind you. So keep that in mind when you assume that the compiler is going to help you out and do smart things. So the instructions are a little bit out of order compared to what you might expect. All right, so I'll go to my prepared screen shot of the code.

So what did it generate this time? All right, so already, this code looks shorter than the previous one, right? A little bit less hairy than that. So what did it do? Well, SIL, once again, is the lower byte of RSI. So it does the compare RCX with RDX, which is actually this comparison. And then it saves that into SIL. And now, later, it does the XOR. So this was the code from before.

And then it does a move of RSI, which contains the same value as SIL, which is CMP's either 1 or 0. It moves that into RAX. And then, later on, it uses RAX for one calculation. And then it also uses the value that's already in RSI. And it subtracts that from register 14, which is where it decided to store the value of B. So that's a much more direct way to either subtract 0 or 1, actually get just 0 or 1, and then use it for a subtraction.

So the end result, and something else to point out, is that the move and subtract commands have a little bit of instruction level parallelism, because right after you set [? LE ?] SIL, as soon as that's done, you can execute both the move and the subtract at the same time, since they use the same source register and put in different destination registers. And the other thing to notice in the comparison is the number of ALU operations. I'll go back to the original one. You see negate, and, XOR, add. So move is not an ALU. It's not an arithmetic operation. But everything else is either XOR, and, add, or subtract, or compare, which all use the ALU.

Now, if you look at the code over here, you have move, move, move, which don't need the ALU. And then the other or four or five need the ALU. But having fewer

ALU op is a good thing.

If you remember from the previous lecture on CPU architecture, the Nehalem, has six execution ports. So it can execute six micro-ops in parallel per CPU clock cycle. Actually, only three of those execution ports have ALUs on them, so that they can actually do XOR, at, and, subtract, add, and so on. The rest of them can do register moves and a couple of other things, but not arithmetic operations. So that's a good thing. So that's an explanation of why the code actually sped up.

Now, just as an overview of what we were able to do, we ran perf stat, the same thing that we did for mergesort, or for the original two mergesorts. And we made a table of the improvements that we made. So originally, quicksort ran in four seconds. And it had an IPC of 0.8 and 11% branch missing. So when we switched from quicksort to standard mergesort, we increased execution time by 20%, which is not good. But the instructions per clock jumped up a little bit. Branch miss rate is still pretty bad.

Now, as soon as we switched it to branchless, we got an 8% improvement over the previous runtime. But note that we were still worse off than quicksort at that point. But the instructions per clock jumped quite a bit from last time, which is kind of what you would expect with reducing the branch misses by a factor of 10. So the CPU isn't spending extra time stalling the pipeline, trying to backtrack on its branch mispredicts.

And now the more interesting thing to me is, if you look at the two silly little compiler optimizations we made, switching from int to long reduced the runtime by 11% over the previous branchless implementation. And then instructions per clock and branch miss both didn't change at this point. At this point, we were just reducing the number of instructions. And then going from there to changing not CMP to 1 minus CMP reduced our runtime by another 7% from before.

So overall, branchless mergesorting is 10.8% faster than quicksorting by the time we were done optimizing, which is a lot better than negative. At this point, we were still behind. And at the end, we ended up ahead of quicksort. And then, overall,

switching to branchless, with all the performance tweaks that we made, was 33.6% better over the branching version. So the non-branching ran a third faster than the branching version, which is pretty cool. And that's something that we learned at the end of the day yesterday.

So to conclude, what did we learn when we were doing this? Well, I think a lot of that applies to the PSETS that you're doing and how you should go about optimizing performance on the code that we give you. And it's profile before you optimize. So before you spend too much time just reading through source code and making wild guesses as to why the code is slow, profile the code. See why it's actually slow. Get some performance counter data. Do some perf annotate, so that it tells you what assembly or what line of code that it's actually spending the most time on.

And then, optimize iteratively. Don't try to do everything at once. Do things one at a time to see whether or not they make a difference. Now, before we arrived at the not, there are so many different ways that you could have expressed not and 1 minus and negative and so on. So if you try them all at once, you really don't know what made the improvement. So you've got to try things one at a time and see what it causes the compiler to do, because you can see in the previous two examples, by the time we made that one optimization, it wasn't just that one assembly instruction that the compiler ended up changing. The compiler, in fact, realized that it could do additional optimizations on top of what we hinted to it. So it did even more optimizations. But the bottom line is, you should optimize iteratively. Try something, and see what effect it has before moving on and trying something else.

And then, as much as I don't like coding assembly, and I don't expect that anybody here codes assembly for the class, looking at the assembly is always a good thing. Just skim through it for your performance bottlenecks. And try to get an idea of what the compiler is asking the CPU to do. And see if there's a way that you can nudge the compiler to do the right thing.

Sometimes we tell you the compiler will optimize certain things, will get rid of dead code, and so on. And most of the time, that's actually true. But it doesn't hurt to

double-check in the assembly and make sure that it's actually doing that. And the output from perf is pretty helpful to doing this. It's not a monumental task of disassembling. It does all of the annotation and interleaving for you.

And finally, learn through practice. That's the only way to learn how to do this performance optimization. You gain a lot of experience from just going out and trying the tools. And Project 2 will have you working with these tools. Project 2.1 does not have a code submission. It's just a write-up, where you get to just play around with the tools on code that we give you, so you can see how to use the tools. And then Project 2.2, which is going out next week, will actually give you a chance to try to optimize code that we give you, using these tools to help you out along the way. So with that said, any questions about--

AUDIENCE: [INAUDIBLE] a write-up. how do you submit write-ups for projects?

GUEST SPEAKER 2: On Stellar. So the handouts for the write-ups are Stellar homework assignments. Just upload a PDF to Stellar.

AUDIENCE: OK.

GUEST SPEAKER 2: Yes?

AUDIENCE: [INAUDIBLE PHRASE]

GUEST SPEAKER 1: Yeah, you just have to build the binary with debug info.

GUEST SPEAKER 2: Yeah, minus G is enough. As long as GDB's break points can see your code, then perf can do it. And it's actually pretty nice in that, even if your code is inlined, it can identify where the inlined portions came from and pull in the proper lines of code for that.

SAMAN AMARASINGHE: I think what's really interesting and important here is not that they found this interesting thing, but the process they went about. A lot of times, when you are coding, you are in control. You have very planned structure how you're going about

doing that. And you follow these through and produce the code you want. And sometimes, some pesky bugs show up. You go through that.

Here is, basically, you basically let the system and data drive what you are doing. And when you start, there's no planned part, saying, I'm going to do A, B, C, D. Basically, you look at the results, and you figure out what might change. And you can go through that.

And I was actually talking with Charles, saying, in my first lecture, I talked about Matrix Multiply. It looks really cool. Every time I do something, it improved by a factor of 2, 50%, stuff like that. What you didn't see is all the things I did that didn't have any impact on the program. You saw this nice tree going down there, all these leaves that end up in dead ends, that had nothing happened.

So if you feel like when you go there, you do a bunch of things, nothing happens, that's normal. Of course, when you tell somebody, you're not going to say, oh, that [UNINTELLIGIBLE]. Say, I did this [UNINTELLIGIBLE], and I did that. And at the end, it looks like these other people are very smart, and they're getting through all these optimizations. No, they spend a lot of time going through these dead end parts. And the interesting thing in here is, one thing is, don't trust anything. [UNINTELLIGIBLE PHRASE] Intel, they have hundreds of engineers working at it for years. And OK, that should be good. Not really.

So a lot of times, it's an end-to-end thing. With performance, it's end to end. So the problem can be at any level. So it could become a compiler, it could be [UNINTELLIGIBLE] network, it can be network, it can be an operating system, it can be in the memory system, it can be in the processor, it can be anything. So if you say, oh, yeah, that shouldn't be the case, probably that's where it is. So all the time, basically, it's interesting.

The nice thing about these kind of tools is that sometimes even the tools might be wrong. So sometimes I have been in situations where you look have to look at the wall clock, OK, wait a minute, does the tool say what I actually see what's happening? Because you get into very skeptical situations. Because a lot of times,

in these kind of situations, the problem can be hidden in many different things. So that's a very good way of going through this. Of course, what you get handed to you, we have set it up in a way that there cannot be too many surprises.

But when you're going through a system and try to optimize and look at performance, this is a really good way of doing it, because I don't think these guys thought, when they started, they'd be looking at compiled bugs or compiler problems. They were thinking about, OK, I'm going to do a little bit of code rewriting and get there. But at the end of the day, what they disclosed was very surprising to them. And it even surprised me today.

GUEST SPEAKER 2: Our plan basically ended at row three, as far as when we first set out. We had no idea was to follow after that. So one thing that I remember is that some students came to us and asked questions, like, yeah, I tried to implement this bit hack here, and it actually slowed the program down. Or I changed my pentomino board representation to this also. But it didn't seem to speed anything up. Well, in those cases, yeah, you need to do more detective work to figure out what happened.

SAMAN AMARASINGHE: So a lot of times, the performance is something like a max of a bunch of different things. So you have done something, that's great. It should give you a huge performance improvement. But something is holding. And a lot of times, you have to do three or four things. And once you do three or four things, suddenly everything starts kicking off.

A good example is loop unroll. You unroll a loop and say, ha, I should get all this parallelism. No, because you have some dependence. So you have to fix that dependence, fix that dependence. Suddenly, you remove everything. Then you get all the parallelism and start running fast. So a lot of these things, it's basically max. You have to get all the things right. And so if you stop halfway through, that means you might be almost there, and you are not up to that. So that's why it just can be a really frustrating process. But it's kind of fun, too, when you get there.