PROFESSOR: So hopefully by the end of the class, we will show you a histogram for the quiz. We are very happy. You guys did really well, so we feel like, actually, you learned something in the quiz, so it makes us happy. We're hoping to have the histogram ready by now, but we don't. By end of the class, hopefully we take a break in a middle, and go through rest of that.

So we are going a little bit off from regular programming. If you look at the class schedule, we are going to have a guest lecture today, but I am the guest lecture, I guess. So I'm going to talk about what compilers can and cannot do because, as you went on last couple of projects, you tried hard to do weird things out of the compiler, create different piece of code, [UNINTELLIGIBLE] programs. And so I'm going to kind of, first, walk through some stuff that is very practical, what the current GCC do or don't do.

OK, so we'll go through some stuff, which is interesting, and then I'm going to-- let's get the outline first. OK, this doesn't work. OK. Then we will talk a little more about where the normal optimizations happen, what are all the possibilities, just very quickly, to give you a feel. It's not all static type compilation. And then go through two things.

One is data-flow analysis and optimization. That's a big part what compilers does. And then also instructions scheduling. Instruction scheduling might not be that important on a superscalar. but it's always good to know what it does because there are some cases where you had to do it in the compiler the hardware cannot do, so we'll look at some of those cases.

So the first thing that we found a lot of you guys did is you try to inline a lot of code

in your projects, and in fact, one way to do that is have you do a macro. A macro basically make sure that definitely get in line, but macros are ugly. There are many things you can't really do with a macro.

The other thing is, we can actually do simple max calculation using function. OK, starting function defined. And the first one is calling this one. Second one is calling this one. So what do you think this is going to do?

How many people think that the code produced between this and this going to be drastically different? Code produced for this function versus this function going to be drastically different? Real different. For some people, [UNINTELLIGIBLE] going to be different. OK, why do you think it's different?

GUEST SPEAKER: So the second function calls the first function, and the first [INAUDIBLE] calls the second one.

PROFESSOR: Yes.

AUDIENCE: So since you have the first one calling max1, which is a macro, the compiler just copied [UNINTELLIGIBLE] out of there. Whereas the second one-- it will try to actually optimize the-- Won't it try to optimize what is inside un64 [INAUDIBLE]?

PROFESSOR: So what he thinks is that is just going to get copied. This will be still a function call, try and do some optimizing. In fact, what it will do is, the first function is going to get inline. Something interesting here, what you see is even though there's a condition here, there's no branch. They have the same old instruction that basically can be used to do a conditional [UNINTELLIGIBLE]. So instead of having a branch and having a pipeline [UNINTELLIGIBLE], this managed to convert this into nice, direct call.

The interesting thing is the second one is also identical, so what it did was it said, hi, I know this function, I can inline. It got inline automatically. You didn't tell you to do, the compiler actually inlined it for you, and then did all the things that are necessary. So what that means is you don't have to write some of these ugly macros and hand

inline. You can have nice functions in there, especially if it's in the same file, it can get inline.

Of course, if you define it to different file, and this file doesn't have access to it, it won't. But if it is the same file, it'll get inline, so you get the same result. So you can still have this nice [UNINTELLIGIBLE]. You don't have to be with optimizations at that level.

So another thing is we have this entire-- Question?

AUDIENCE: On the previous slide, other than looking at the assembly code, how do we know when the compiler based this on?

PROFESSOR: You've got the assembly code. Question?

AUDIENCE: Why do you prefer static converge versus inline?

PROFESSOR: So the reason I did static is basically, I want to make sure it's not visible outside the file, so if you don't make it static, what'll happen is everybody else had the visibility and then you kind of pollute the space with a lot of names. So if you give static, it's only within you. If I had inline I can't ask it to forcefully do that, but what I'm showing is you don't even have to say inline. It'll inline by itself. Question?

AUDIENCE: [INAUDIBLE] this function to be inline without actually [INAUDIBLE]?

PROFESSOR: Actually, gprof, what'll happen is the file will vanish from gprof, isn't it? The function will vanish because gprof will basically, if you have a function, you go look, you don't see the function. And it might give, even, a bad impression that OK, that function is not important, so that you had to be careful in that because when you look at gprof, you will see some files. The inline find some functions, inline function won't be there.

Am I right, or is it doing anything interesting to the samples? No. [UNINTELLIGIBLE] vanish and oh, yeah, that function is not important, but in fact, it might be really important, but it got inlined. So that's one way to, a little bit, worry about. Does it make sense? OK.

So we learned bithacks. So it was really fun. We are learning all these interesting bithacks, but the interesting thing is, in fact, GCC compiler also knows a lot of bithacks, so it's also a pretty smart compiler. So if you have something like this, what do you think the smd would be?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, it actually did this one. This is very interesting because what it did was it didn't do shift. It did this load effective address quad instruction, leaq instruction.

What it does is multiply these two, and it's add. It does nothing in here to add, and then it can add a constanant. So this very complex address mode that is being used for completely different purpose. So this [UNINTELLIGIBLE] address.

It's doing this multiplied by this, there's nothing to add, there's no offset to add, save it here. So it's actually doing this multiply-by. The nice thing is it doesn't affect any things like condition codes and stuff like that, so this is a fast thing to do. OK?

Actually, that's interesting. How would [UNINTELLIGIBLE] 43? That makes it a little bit more complicated. What do you think? How to multiply something by 43? Anybody want to take a guess [UNINTELLIGIBLE] multiply 43 [UNINTELLIGIBLE] mul43?

AUDIENCE: [INAUDIBLE] multiple by 32, have that multiplied by--

PROFESSOR: So it did something interesting. So here's what it did. I want you guys to stare at it a little bit and see if you can even figure out what's going on here because this is kind of funky.

So what happens after the first leaq? What's an rax? What's an rax after first leaq instruction? Anyone take a wild guess?

AUDIENCE: Five?

PROFESSOR: Five. Yes, exactly. What it does is it's 4a+a because this is a+a, 5a in here. And then after here, what happened? 4 times this one plus this one.

Yeah, it's 21 because 4 times this is 20, and you add the whole original rax 21 here, 21. And then you do this time again 42, and add another one, 43. So it did all this interesting three instructions to get to 43. And so this is fun. You can spend days giving different meaning to this and see what the compiler generates.

And I notice at some point say, when do it give up? And it doesn't give up for a while. It start creating some crazy things. OK, try one more thing.

OK, this has to be hard. 255. How did it get 254? There's no easy way to do that. How did that to that? This is the very interesting thing in here, so I will show you this one.

So here is that instructions you generated. So what it did was, it's doing here, getting it 2a, by doing leaq because since it didn't give a multiply, it just multiplied by 1, so it's just adding these two. You get 2a. And then it multiplied by 128 by doing a bitshift here, and then it got such a wonderful thing. Why did it do this instead of doing one instruction to 256?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, then it went and subtracted a 2a again, and got 254. So it went overshoot, subtract, and subject one more, so [UNINTELLIGIBLE] got calculated here. It does a really smart way, I don't know what complex logic is there, to basically figure out these combinations of instructions that can do multiplication.

We planned on a bunch of things until we run into a couple of thousand. It was doing these weird patterns after a couple of thousand. Especially if it is to close to 2 to the power, it easily find it. Even primes, it managed to find.

Actually, before we gave a prime, and it found it because it found the closest thing, and a couple of things [UNINTELLIGIBLE], and you can get to the prime. So it's kind of interesting how find goes in just [UNINTELLIGIBLE] multiplies.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: So this is a very interesting question. So what it's doing is, it has realized somehow that doing a direct multiply by 254 is going to be slower, so the multiply instruction-- if you go, I think you can also look at multiply instruction, how many cycles it's going to take to come--

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Oh, because it needs to get this 2a again. So it's [UNINTELLIGIBLE], use the two 2a. So by calculating that, it kept it. I don't know, you might be right, because if it [UNINTELLIGIBLE] 2a to 6n2, then there's no dependency, and right now, there's a dependent change here.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Twice, yes, something like that. Or calculate this separately. 2*8 and 256, and then add and subtract them. Might be interesting. So there are other ways of doing that, so in fact--

I don't know why, it might be in because [UNINTELLIGIBLE]. I don't know why they didn't do that. But it's thinking. It's thinking, look at all instructions, sequence, how long it would take, and it find this interesting sequence.

So sometimes when you are looking into optimized code, they will look like something crazy that you can't read because it does things like this. So you found a simple multiply [UNINTELLIGIBLE], and end up with a piece of code like this. And so you need to kind of decipher, seeing what's going on backward. So that's why reading assembly is sometimes hard, especially optimized assembly.

OK, so I did absolute value, and look, it did the bithack we basically learned in class. You can go look at that. This is the entire thing that Charles talked about to find absolute value. It knew that, so it has attended Charles' lecture. That [UNINTELLIGIBLE] programmer.

OK, so here's interesting thing. So what I did was I am doing update, and I have this big large array here, and I'm checking the index to be within 0 and this value to

before I updated, because I don't want to write out the bounds on this setting. OK? Makes sense, because I want to make sure that I write in the bound.

Interesting thing here is I am doing two checks, here. I end up doing only one check here. What happened to my other check?

AUDIENCE: [INAUDIBLE], how big was the array, and [INAUDIBLE].

PROFESSOR: No, no, [UNINTELLIGIBLE] something a lot more simpler. So to give you a hint, this is unsigned value, and I'm doing unsigned compare. What happens if the value is more than 0? A signed value that's smaller than 0 is what it is like in unsigned. It's a huge number. It [UNINTELLIGIBLE] to be bigger than this one, so because of that, it can just say, OK look, I don't have to check that. I can unsigned compare, and I will get anything less than zero also in there.

So one more thing. So before I continue with this one, so if I actually put it in a loop and say I'm going to trade [UNINTELLIGIBLE] to this value in here. Then what it'll do is, at that point, it'll inline this. And when it can complete [UNINTELLIGIBLE] near the check, it will know that, in fact, my bound is going from 0 to this one, so I don't have to check the bound.

So what this did was this inlined this function in here and completely get rid of the checks completely, and this is basically the branch condition in here because it said, OK, look. These things are redundant because I know because I'm trading from this to this value within these bounds [UNINTELLIGIBLE]. So it is smart in that.

Do you see this, how this is going? Cool stuff the compiler does. So this is why compilers are smart, when they are smart.

And the interesting thing is, here's another one. So now see [UNINTELLIGIBLE] imagine, because less than 0, I can do that. How about if I'm checking from 5,000?

So we generated this funky code. It subtracted a 6 here in the value, and then checked for [UNINTELLIGIBLE] because it kind of shifted the value to a 0 basis, basically. And then you can check that thing, and then can basically get two

conditions down to one.

See, the thing is there are many places where bound checks is very important. If you are doing a lot of adding compilation stuff like that, if you don't want to have buffer overflows and stuff, you want it with bound checks. And so optimizing bound checks is a very important thing. So having these kind of things can, in many programs, probably give good performance, so that's why compilers are really good at it and spend time trying to do bound checks. So this is kind of interesting way of doing that.

So the next thing I want to look at is vectorization because all these machines we have have this as the instructions, that can run really fast, and you probably saw it in there, and see what kind of code will get produced after doing something like that. So here's a simple program.

So I have two arrays, and I'm just copying A to B, something very simple. And also the other thing to notice, I know exactly from where to where I'm copying, and I also know which arrays I'm copying, so when you look at it, it produces a code like this. So what it's doing is it's basically making eax0 here by doing xorl. And then basically, moving the value A into the xmm registers, much larger. Instead of having to [UNINTELLIGIBLE] 16 of them, and copying it back into B.

So basically, you are doing copying here, an increment by 16. By now, every refresh, you're coping 16 of them. Why could I just be done with just putting this small piece of code? What additional information this is taking advantage of?

AUDIENCE: Does it know that [INAUDIBLE]?

PROFESSOR: Exactly, because it knows that it goes from 0 to this value. In fact, it knows it's a multiple of 16. So it knows that. That's why it do that.

It knows exactly, and these things are nicely aligned to the boundaries, word boundaries. So it knows that. So I can read that, and I know all those facts, and that is why I can do this computation.

So now, you start doing that, did one simple change. You start going from value, I went to end. 0 to end. I know where it starts, and I know where it's ending. Ending is somewhere at N. I don't know where the end is.

Then this has to do something a little bit difficult. So the code produced looks like this because, now, I'm not going to go through this code. The only thing to say, this is actually doing still a memx instruction, but its trying to make sure that because N it might not be a multiple of 16. You have to take care of the final number of iterations outside that, so you had to go up to the multiple, and then basically do a normal loop one at a time. So as we produce a little bit of a more complicated piece like that.

So that's [UNINTELLIGIBLE] compiler has to do. And so then you have a piece of code like this. The interesting thing here is, now, I created basically a function, where it's not A and B. I'm giving two arrays as arguments, and then I'm giving a size to copy, and I'm copying that. And I would have an extremely complicated thing that's getting generated.

Why is it complicated? What do I have to know, when I get this function, to make sure that, first of all, it's still doing xmm somewhere in here. What that means is it's trying to do this very fast copy of a multiple using [UNINTELLIGIBLE] instruction.

But what else can happen in this function? Because compilers delete all the cases. What are other cases tests deal with?

AUDIENCE: May not be aligned.

PROFESSOR: May not be aligned because, for example, because xmm assumes that they had the word boundaries. When you read 16 bytes, we assume it's aligned with 16-byte boundary. It might not be aligned, so you have no idea where these two are coming from. So that's one thing is they might not be aligned. What else?

AUDIENCE: They don't even have to be a [INAUDIBLE]. Because I mean, that thing could just be copying up to N. But it might just be copying partially parts of the array.

PROFESSOR: Yes, yeah, that's true. So what that means is because arrays [UNINTELLIGIBLE] it, but arrays is somewhere in memory, just you do two-point to starting point. That is what x and y are there, two starting points in main memory, and start copying there.

So what else can happen because of that? Because in A and B, we knew they were two separate arrays. What else can happen? Back there.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes. So arrays can start to overlap, and if arrays are overlapping, then you might end up in an interesting situation. So you have to figure out if that arrays are overlapping, whether they're aligned. Actually, there are two types of aligned.

One is self-aligning, so assume we took these arrays and start copying from the byte 3. So then what you know is basically byte 3 to 16, it's not aligned. You can't start copying chunks in there. So you run bytes, but when you run up to 13 iterations, then you end up in, again, aligned chunks.

So in that case, you just run the sum preamble to first aligned place, and then you go to aligned chunk. But if this is starting to at 3, this is starting at 8, then they're never going to be aligned. The things are copying A and B are not aligned, so then you have to treat it differently.

So there's a lot of different cases you have to do, so if you just give something like this, the problem is the compiler has to deal with these thousands of different cases. And in this one, since this small, it probably tore through all the possible cases at the [UNINTELLIGIBLE]. So it's dealing with all those, and checking over everything and trying to find optimal case and do that fast, hoping that you get optimal case for it. But if you have something more complicated, the compiler won't be able to do all of those things, so it might give up. So the interesting thing to here note is more information to go into the compiler is better. And then here, compiler has to divide a lot of things, but probably not happen.

Another interesting thing is now, the first time I just copy to where is A to B, in memcpy4, I just call memcpy3 by doing the same thing, A to B, copy 1024

[UNINTELLIGIBLE]. This is the beauty of inlining. So what it did was, in memcpy4, it inline memcpy3 and substitute X and Y to A and B and end 2,024. And it realized that it doesn't have to do all these tests like it did.

What it generated is very close to what we got here because after inlining, it should realize, wait a minute, I'm copying A to B. I know we have the start. I know we have the end. I know the size.

I know all of these things, and I don't have to do any of these things. I can actually generate this very simple piece of code. So I think that is a neat thing.

What this shows you is, in fact, if you can build this general function of things in there, and then you can call them, and if it is done right, the inlining will basically do all optimizations. So you don't have to have 50 different memcpies for all the different things in your code. If you wrote a general function, and you call it in a way it can get inline and got that as efficient as possible as hand optimization.

I think it's a real interesting thing, and what does for you, when you're doing projects, you don't have to do all of these very complex and small functions, hand inline stuff like that. But it's always good to check that, in fact, the compiler's doing that because you don't know. You assume the compiler's doing that, and there might be cases it might not be. And I will show you one example here.

So I want you guys to look at this function a little bit. OK? I am doing two memcpies. I am copying 1,024 elements.

One, I'm doing ai+1, a into-- this is XY, this is X get copied into Y. ai+1 to A, so that means I have array like that, array like this. I am giving ai+1 as the source. I am giving this as the source, and I'm doing this as the destination. OK, what does this copy do?

I'm copying 1,024, yes. So the first one, this one gets copied to here. Second iteration, this will get copied. Third iteration, this will get copied to here. What does it do? Yeah, I just do one left-shift of the array.

My second example. I give this as my first element. This as my source. This as my destination. What happens here?

AUDIENCE: All your copies have the same number.

PROFESSOR: Exactly. All of them will copy the same number. OK, so now, if you look at the code that's been produced, so the interesting thing here is it realizes, in this one, I can still do mmx because I can still copy, take a chunk, and copy it one back, take a chunk and copy it one back, take a chunk and copy it one back. OK, do you see that?

But what does the next one do? [UNINTELLIGIBLE] mmx is, and what does this one do? Copying something from dl--

[? movzdl ?] array expressed dl. Reverse dl, is this copied, this one? I hope I copied it properly. That doesn't look right to me.

So this is interesting. So I might have missed [UNINTELLIGIBLE]. I think it takes-- This doesn't look right, does it?

AUDIENCE: So what's a bound? It's char, I see. One byte.

PROFESSOR: Yeah, one byte.

AUDIENCE: So what this is doing is just taking the first byte, and then just moving it.

PROFESSOR: Into edx?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oh, right, this is actually doing the right thing. It's doing the right thing, so what it does is this move this one into edx, entire thing, and then this calls the dl, it gets the first byte out of edx. Do you see what's happening here? So a gets into-- first [UNINTELLIGIBLE] the first byte out in here, and then you keep copying that byte one at a time into this location.

AUDIENCE: So it's not smart enough to [INAUDIBLE].

PROFESSOR: So this is where we find something interesting in the compiler.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: So what doing is, in here, you are copying this byte into edx. So dl is the first bite out of edl. That's byte, because what--

AUDIENCE: The first byte.

PROFESSOR: Yes, because except the six address themes, do you do r32e, r64e, 32, and just dl is just first byte [UNINTELLIGIBLE], but [UNINTELLIGIBLE] low byte of that. d is just the higher byte.

AUDIENCE: So why does it just take the first byte?

PROFESSOR: Because this byte get copied into everything here. So do you see that? This byte is the one that-- because that's what happened when this copy is basically this byte got everything [UNINTELLIGIBLE] got replaced by this first byte in here. And you incorporate in there, and then it just goes around copying it in here.

So I try, I did it this way, so what I did was basically went from 1 to 1,025. [UNINTELLIGIBLE] 0. This is basically what happened. So one 2,025, I got a 0, and then basically that's what happens in here, same thing here. OK? Do you see what's going on?

But I just did something else, so assume [UNINTELLIGIBLE] is doing right here. What happens if I just use something else? I use B[0], so it should be the same, isn't it? If I use b[0] here, instead of doing A[i], this should be B[i], isn't it?

AUDIENCE: I'm sorry, where is [INAUDIBLE]?

PROFESSOR: Yeah. It's a different array. Sorry, I didn't put it here. It's a different array. So instead of A[0] here, I put another array, B[0], here.

Does it matter whether that's A[0] or B[0]? B[0] is a different array. It shouldn't matter because a different array, different element that you don't do that, but the

interesting thing is if you do that, it managed to convert it into mmx. So this is where the compiler is basically falling short a little bit because it could have done this for these two.

OK, it's the same thing because what it does is it takes this one element from me and copy it everywhere. I could have done it, but for some reason, the compiler decided if using a different array, B[0]. I can do it, but I'm doing A[0] [UNINTELLIGIBLE], even though these two are basically identical except these are different. I'm not doing any kind of [UNINTELLIGIBLE], anything. Question?

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Yes, so what does is when it goes somewhere, it's doing pac. [UNINTELLIGIBLE] pac instructions in here. What it does is it takes a byte and kind of makes multiple copies of the byte and create a larger copy than the 16 copies in there, and then it can kind of stamp it everywhere.

AUDIENCE: [INAUDIBLE] because the A[0] would go off of A[1].

PROFESSOR: So what it's doing is copying A[0] multiple times one at a times slowly. So what this does is it takes B[0], make 16 copies in there, in registers, not in memory. I created a template of 16, and I kind of stamp it as I go.

AUDIENCE: That's probably failure of its alias analysis.

PROFESSOR: Yeah, it's failure [UNINTELLIGIBLE]. So this is where the compiler does some magic, kind of very complex things in here. But somewhere in the compiler it failed. so what you want in here is a code looking like this, but it produces this one.

So this is where the compilers are great. It do some amazing things, but it's not infallible. It can do, there might be corner cases that data analysis fails. So that's why it's always good, even though you can take advantage of the compiler, to look at what's generating.

And sometimes when you tweak around it, you suddenly realize, wait a minute. I can get the compiler do something better, and then you can say wait a minute, now

how do I work myself back? Sometimes, you end up changing your SQL a little bit like the examples, the TAs showed when they were doing their demo, that if you tweak it a little, you can actually get the compiler to do that, instead of trying to do these things by hand. So the compilers are powerful, but you have to be careful.

Another interesting thing is this factorial here. So normal factorial is basically you call a function call with x-1 and multiply by x. But you know functions calls are very expensive, and in fact, GCC knows that, too. So what GCC did, it basically eliminated the function call and converted it into its [UNINTELLIGIBLE] function here.

So if EDA got x in here, and then it goes to a loop, so it first check with [UNINTELLIGIBLE]. If it is one, you go to the end and return it. You are done if x is 1 or less than 1.

And otherwise, it goes through a loop. It doesn't go do any function call. It just basically calculate this fact value inside EAX and keep multiplying [UNINTELLIGIBLE]. So it can take this simple recursive functions, and also convert it into [UNINTELLIGIBLE].

So it does some very, very fancy stuff in the compiler. So the compilers are fun when they work. But the key thing is there are many cases it doesn't work.

So next, I want to switch gears. Any questions so far? Sometimes it's fun to find breaking points in the compiler.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: If I use, in some-- this is not static?

AUDIENCE: Yeah.

PROFESSOR: No, it won't make a difference because what it says is if it is not static, it's visible to the outside world, but within this function, it's the same. So what it does, it kind of like limiting my pollution because otherwise what happens is everybody outside will

see these names. So if you use that name again somewhere, it might just use this one.

[UNINTELLIGIBLE PHRASE] this is within the file, nobody else should see this. It's creating a local copy. It's kind of a poor man's class heierarchy.

In Java, basically, each file is a single class, and you make sure that things inside the class is not visible to outside. When you make static, you made it only visible within that file, so you kind of make [UNINTELLIGIBLE]. You can think about your file as your class, and so you limit the scope of the variable doing that. So some benefits of object-orientedness can be [UNINTELLIGIBLE], I guess. [UNINTELLIGIBLE] static variable, it's not a class variable.

OK, so next, before I get into doing compilers and say what compilers do, I want to give you a [UNINTELLIGIBLE]. There are many different places where you can do optimization. So if you look at what happens in the program, program first goes through compile time, compiles each file, then it links all the files together. At some point, the files will get loaded into your machine, and then it'll be running. So if you load things in a compiler, you have full access to source code, it's very easy to kind of look at the high-level transformation, low-level transformation, you can look at the entire gamut of things to do.

And the nice thing about compilers, compilers can be slow. Nobody's going to complain. It's not going to be part of your run-time, so you just would wait, but it's you, not the customer.

But the problem with compilers, it doesn't see the whole programs. You see a file at a time, so all this inline things and stuff has to be in the file. You can't put in a different file and get [UNINTELLIGIBLE].

And also don't know the run-time conditions because that's run-time. It might be having different inputs, different size of load and stuff, that I don't know any of those things. And also, I don't know about the architecture, so if my compiler have to make sure that it works on AMD machines, Intel machines, stuff like that, of course,

you can use special flags and try to comply for one machine, and breaks, you finish on the other one. But you don't want to do that, so the compiler has to be a lot more general, and this can be sometimes problematic.

So when you're going to link, the nice thing is that this is a place you have the entire program available. Sometimes, people try to do things like inlining in the linktime, because that means you know everything in there, so you went with couple different file I can inline it because I have access through here. And still, there might be things that's not available, like dynamically-loaded classes and dynamic-loaded data, and so things like Java might not be available. And of course, you don't have access to source most of the time.

AUDIENCE: Sorry, sir. What do you mean [INAUDIBLE]? Do you have the full program [INAUDIBLE]? But so how do you say that [INAUDIBLE]?

PROFESSOR: So dynamic links, if you have something like Java, there might be some data that kind of get dynamically generated or dynamically linked. So when you're running, if you're running right [UNINTELLIGIBLE] your browser, all those Javascript classes and stuff like that, you don't have access to because those are coming in here. So there might be places, things that it gets as it runs. Not in C, but in other languages.

And the load is interesting time. Here, load time is important because when you double-click, you want your program to appear fast. You don't want it to take a long time. But you have kind of access to all that code in here, and you have some idea about the run-time, also, the architecture, and stuff like that, what you have, not the run-time, but the architecture, exactly what machines you are running. And then, of course, you can do it run-time.

The thing about run-time is you have full knowledge of everything, it's great, but every clock cycle you spend optimizing is one clock cycle you take away from the program. So things like Java JIT compilers, they try to do minimal things, so very fast things. It can't do a lot of complicated things because it's too expensive.

OK, so we're not talking about any of these things any more, but it's always good to

know, as you go about using Python or Java or JavaScript and stuff like this where is this thing happening to my code? Because it might not be all compile-time stuff. It might be happening at different stages. So you need to know who's actually mucking with your code, and know that there are other people who can muck with your code.

So next, I want to switch into dataflow analysis. So this is what compilers are good at, and compilers try to do all the time. So it's basically compile-time reasoning about run-time values and variables, or expressions, within the program at different program points.

OK, so that means compile-time, I need to know I have this program point, what could it be. So things like which assignment statement produced a value or variable that I am using? OK, if I use a value, who actually created that value?

Or which variable contain values that are no longer being used by somebody here? So that means I am trying to analyze the program and watch the range of values that each variable can have. So the key thing here is this has to be true for every possible input at every possible execution. Normally, [UNINTELLIGIBLE], and this time, I know why my variable [UNINTELLIGIBLE], but every possible time, this has to be true.

OK, if there's a condition that something can happen, you have to make sure that condition is not going to break your program. Last thing you want from optimizer is to basically start producing different results. Even [UNINTELLIGIBLE], it's not good, so you want a compile optimizer to kind of produce the same result that you got without optimizing. And this is why this has to be [UNINTELLIGIBLE].

So first, I want to go through a little bit of example, what kind of things the compiler do. You probably have seen this in one of the earlier lectures. We talked about some of this as hand optimizations, but I'm going to go through some of them by using this program.

It doesn't mean anything what I'm doing here. I have a loop here. I'm calculating

some function in here. And then I am adding something else to this x here, and I have some initializations in here, just something that I can demonstrate what it does. So it has no meaning for this one.

And here's the assembly instructions. I'm not going to go through assembly, but [UNINTELLIGIBLE] you can actually create and understand why this is happening in [INAUDIBLE]. So I [UNINTELLIGIBLE] into two slides.

The first thing you can do is think of constant propagation. So what it says is for all possible executions, if a value that has in a variable is the same, and we know that value, that's a constant. And I don't have to keep that value in that variable. I can replace that with a constant.

Sometimes, when you look at dataflow optimization, you can say this is done. As a programmer, I will never do that. This is something you should be doing, for example, have things like constant variables that constant values or lower.

But sometimes, something looks dumb, but what happens is sometimes when you're in optimization does, one optimization might lead to code that looks like. That can lead to it. I will show you something sometimes that you might not find a code that looks dumb, but previous optimization will leave, or change the code in a way that this optimization can take advantage of. So nice thing about this is you don't need to keep values in the variables because you can free some variable, that means free RAM registers. Also, most of the time when you do constant propagation it leads to [UNINTELLIGIBLE] optimizations.

So in this program what are the things that can be constant propagated? So we know x equals 0, x's are constant up to this point. But since x get modified here, my dataflow say wait a minute, I am going through this loop, and x is constant from here to here. But after this point, x is not constant because it get modified in here. So that's what dataflow is going to say, and so I can't do that x.

But [UNINTELLIGIBLE] why it become constant here? All input that goes into this loop, has to go through here, becomes constant in every path. And then it doesn't

get modified in this loop at all. OK, so then I can actually, through constant propagation, get to the file. OK, so now I have a program like that.

So normal compiler optimization is done by pass-by-pass. A lot of passes get repeated multiple times, so I leave it like this. So even though this is just simple thing, but we leave it to somebody else to optimize that, which is what we call algebraic simplification. Basically, it says you go to your, whatever, fourth grade, fifth grade, sixth grade algebra book-- I don't know where you learn, somewhere you learned algebraic --and they have all these very simple rules, like something multiplied by 0 is 0, multiply 1 by that, and all of those rules, and then you can just busy code them up and look for these patterns and replace. And that's what the compiler does.

And in fact, we look at something like this, a simple shape, but you saw before that, it do much more complicated things. And it's a lot less work at run-time, and also it leads to more optimization, so it can simplify things in here. And other thing is, sometimes instead of algebraic simplification, kind of weird things. If you want exact precise, for example if you're doing floating point, because floating point, a plus b plus c, is not b plus c plus a, are different because you can get small teeny differences in these kind of-- [UNINTELLIGIBLE] and associate duty, and some people care. Most people don't because it's so small, most people, they don't care. Others do.

And also sometimes when you do this optimization, things like overflow and underflow, that happens because if I do x plus x minus x, or x is very large, x plus x minus overflow, and then you end of doing minus x because it overflows. But instead of x plus x minus x, it's just x, you don't overflow anymore. So you have changed the behavior of the program, but most of the time, compilers think that things like that are special cases. They are not the normal behavior, so changing them is probably OK. Sometimes, you can't do anything.

So now here, what are algebraic simplification I can do? What can I do here? Yeah, I multiply by 0, [UNINTELLIGIBLE] this, that. At 0, I leave it here, and then there's

another algebraic simplification, I can do that, but now, I am leaving it here because there's no algebraic simplification.

X equals x is-- there's nothing you can do. That's called copy propagation. Copy propagation says you're just making a copy of one value to another, just get another copy.

You don't need to do a copy. Very simple thing in here. Less instructions, less memory registers because we are not copying. However, when we [UNINTELLIGIBLE] register location, I will talk, basically. If I use the same register now, I might have things that was in two registers, x copied to y, now it's all in x. So that means I might have some variable in the register that you call my interference graph.

I'll talk about this in a little while, so I'm just forward referencing. That might not be easily register locatable. And so in here, x equals x. I can get rid of that.

And another interesting thing is common subexpression elimination. If you do the same thing multiple times, you calculate it once, less computation, Cons is you need to keep this result somewhere between the two users. So if I have too many of these things, I might just run out of registers to keep these values calculated.

And also interesting thing is, this can hinder things like parallelization. When we get there, we can see that by adding additional dependencies in there. So in here, what are the common expressions? Either you guys are bored, or this slide is way too hard. You're bored?

AUDIENCE: [INAUDIBLE].

PROFESSOR: y plus 1, OK, good. So there's y plus 1 in here, and I can calculate it once, and then I can just do the multiplication of that, and do that, and voila. It got rid of two addition and one multiplication to one addition and one multiplication.

OK, next thing is dead code elimination. So if you're doing something that nobody's using the value, why do you do it? And less computation, and maybe you release

storage because you're not storing these values your computing, and that's really nice.

And there's not much of bad things about dead code. Dead code is pretty dead. You can get rid of it. So here, what are the dead code you have? I want keep you at least somewhat engaged, so see if you can find my dead code.

AUDIENCE: y.

PROFESSOR: y, yeah. I got rid of [UNINTELLIGIBLE]. Now, I don't need it, I can just get rid of that, and then I can even get rid of allocating y. So I got rid of both instruction and some memory-allocated registry that used to keep that value there.

Another interesting thing you can do is loop invariant code [UNINTELLIGIBLE] because loops are very important. Most of execution time is mainly inside loops, so if you can get something out of a loop, that's really good. We talked about that previously. But you have to worry about, basically, two things.

One thing is that when you move too many things out of the loops, you have to keep all those values in registers, so that means you need more registers inside the loop. Second thing is when you execute that, you have to make sure that it have the same behavior as when you run the program. How about special cases, the loop never get executed.

First let's look at this. What other loop invariant expressions in here?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Hm?

AUDIENCE: 4 times [INAUDIBLE]--

PROFESSOR: 4 times eta a divided by b. OK, good, I just moved up there. So I did that, but why am I really wrong? Why won't the compiler do this? Give me a case that this would change the program behavior.

AUDIENCE: [INAUDIBLE].

PROFESSOR: 4 times overflow, yeah, that can happen. That's one case, but there's something that can happen-- overflow happens in very large numbers, people don't care that much, but there's something that can happen a lot more.

AUDIENCE: If B is 0, and then N is less than 0?

PROFESSOR: Exactly, when B[0] and N is less than 0. I am going to have a divide by 0 error in here because I am going here, dividing by 0. That would have never happened because the loop wouldn't have gone and executed it.

So normally, when you do things like that in a loop, the compiler generate a place called a landing pad, which basically is, before you enter the loop, you check whether the loop will ever get executed. And then go to the landing pad, and then go to the loop. So the landing pad will be run only when the loop at least has one iteration running, and so you can move all those thing in the landing pad.

So here, you can see there's no landing pad. The code generated probably would have, and so I did something that is, you would see in fact, the optimized code I did didn't do that. So GCC minus [UNINTELLIGIBLE] is smart enough not to do this.

So then there's another type of strength reduction, which is saying if I go something like a times i, what I can do is just, instead of doing a times i, I can basically make the first iteration initialize it, and every time you can update the previous value. OK, so array times i, the first it's 0 and next time it'll be t plus 80 plus this, so I can keep updating that. OK, so this is really good. I have this computation because now we just sort of multiply, I just made it add. But I have a lot of problems that can happen here.

First of all, I have, now, this one. I didn't have to keep this value anywhere, only when I needed it. In here, this value has to be varied through out the entire loop because I keep updating that value, so I created another need for a register. Before now, I only needed it at that point. I could've [UNINTELLIGIBLE], rarely used it, but now it just to be there throughout the program I created in there.

Also what I fear is what they call a loop-carried dependence. Every time you run iteration, you use the previous iteration's value. When we go into a parallelizing loop, you suddenly realize that means I can't run them parallel, so this creates a huge problem in parallelization. So you do [UNINTELLIGIBLE], strength increase, when you go to parallelize. And you can undo these things.

So in here, one thing you can do is you look at something like u times i and say wait a minute, I don't have to multiply by i because it [UNINTELLIGIBLE] 0 like this, and I can just allocate a value b and keep it updating by v, and then I did that, allocated a variable in here, allocated 0. And [UNINTELLIGIBLE] this, I just basically put v times 0. You see that? I just basically got rid of a multiplication and convert it into addition, but I paid some cost by, I need now this additional register that is true all throughout the entire thing. [UNINTELLIGIBLE] I just calculated that expression.

And the big thing a lot you get performances register allocation, so most processes have very few registers. In fact, one big change that when you went from [UNINTELLIGIBLE] is to get additional registers. Registers are very important.

I will go through register location a little bit. So what happens is when you have a program, you have a control goes like this. So this control going, executing something that defines this variable x, defines variable y. And here, you use variable x and variable y, and there are different paths the program can go through. There are two paths can merge in here, here, you can expand in here.

So this is kind of the flow of the program in a small part. So what you can say is this definition [UNINTELLIGIBLE] here, so this value, this line in between here-- because you can't get rid of it. When you decided you had to keep it somewhere because somebody's going to [UNINTELLIGIBLE]. And this definition is used here, so when you decided you had to be [UNINTELLIGIBLE] in here. When you [UNINTELLIGIBLE] is only used here. Nobody uses here, so this has to be [UNINTELLIGIBLE].

Interesting thing about x is there are two definitions of x that might be used here,

and this definition might be used here or here. So you put all this into what they call a one web because these two definitions might-- either one of them will be used here. This definition will be used, either one, over here, so this value has to be [UNINTELLIGIBLE] in here, kept somewhere.

So then what we say is we give names to these, so this is s1, s2. Somebody has to keep this value. s2 keeps this value, s3 keeps this value, s4 keeps this value.

The interesting thing is how many registers you need to keep all those values. That's why the entire thing of register allocation. So what you do is this really cute mapping of this into nice theoretical problem. So what you can say is each of these regions, we make it vertex of a graph. If these regions overlap, then we get edge.

S1 and s2 overlap. That means you can't use the same register to keep s1 one and s2 because before s1 is finished using, s2 has to be free of that. OK, there's overlap in here, so we've created edge in here.

OK, s2 and s3. So s2 and s3 overlaps here because at this point, both value s2 and s3 has to be kept. So I create an edge in here.

OK, so I create an edge. Every time I say those do values need separate registers. I can't keep the same register.

And of course, s3 and s4. s3 is here. s4 can be in the same register. So there's no edge here.

os1 and s4 can be in the same register. os2 and s4 can be in the same register because they are not live at the same time. They are live at a different time of the program execution.

Now, what you can do is, you have a graph, you have edges, and there's this very famous problem called graph coloring problem. How many heard of graph coloring problem? OK, good. So what happens is now we can figure out how many colors need to color this graph, and that is the number of colors of the number of registers you need.

So if you have a graph like this with no edges, you can color it with one color. How many colors for this one? Two colors. How many colors for this one? People said two colors. Yes, you can color it with two colors. How many colors for this one?

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's three-color [UNINTELLIGIBLE]. So there's all these algorithms [UNINTELLIGIBLE] and say no. You can see by coloring this how many registers I need. And the interesting is, if you need more colors than the register you have, that means you can't register allocate, and at that point, you need too many things to keep. You don't have that many registers and that [UNINTELLIGIBLE].

That means you take edge and say, ah-hah, I can't keep both of these guys in the same. I will take some vertex out and say this vertex can't be in there because I can't put it into register, and I spill this out. And you can re-color the graphs.

You can spill it, and of course, spilling is costly because now [UNINTELLIGIBLE] value in the register, it's in the memory, so every time you need it, you had to bring it back, send it back, so it's going to be expensive. The nice thing is to see how much you can keep in the register.

So I have enough registers for this program, so I found registers for all these things instead of putting it in memory. And now, this is [UNINTELLIGIBLE] register allocation in a pseudo C code, so this is the kind of optimized code, and this is the generated-- Basically, all four of the original program generated.

But in here, I move this one up. But in this one, actually, the division didn't get moved up, so [UNINTELLIGIBLE] actually inside the loop because it's realized you can't do that. But interestingly moved the multiplication out, so that it didn't care about the overflow. It says, hey, overflow, it can have an overflow, but it will worry more about divide by 0. OK? Any questions so far?

So here's the optimized code, and if you run it, there's seconds versus 54 seconds. Just GCC [UNINTELLIGIBLE] 0, GCC os, so it'll produce very compact optimized

code. So the key thing is what's [UNINTELLIGIBLE] these optimizations.

The key thing is you have to guarantee, when you optimize, all that these programs [UNINTELLIGIBLE] from unoptimized, optimized, all the valid input, all the valid execution, and all valid architecture that you're supposed to run, you can't do the same thing. Otherwise, it's not a good optimizer if it does different things to code. So there are a lot of things that means you have to be very conservative in [UNINTELLIGIBLE] cases. So you have to understand both control-flow and data accesses, and make sure that you understand them, and if any of them, the compile-time analysis cannot understand, the compiler give up very fast.

So the thing is, most of the time if that information is not available, compilers reduce the scope of the region [UNINTELLIGIBLE] the transformation. So we have this point, I don't know beyond that. I can only do a small amount of transformations here.

Or reduce the aggressiveness of transformations, and sometimes just completely leave code alone as it is because it couldn't, even the things you know, no sane program would do, and of course, your code will never do. The compiler assume, if it is a valid C semantics, it might happen. Even though some of them looked really crazy. If it is a valid possible way of doing it, compiler has to worry about it, and not do that. So it's here to be careful of that.

So first of all, control-flow. That means it doesn't work on possible paths of the program when you execute that. And the way you look at this, you can add this call graphs in the high-level [UNINTELLIGIBLE] the call in here, and control-flow graphs within the metadata function how control goes from.

And what makes it hard for compiler to analysis this? Bunch of things [UNINTELLIGIBLE] function pointers. You probably haven't done function pointers, but if you have function pointers in the compiler concepts, I don't know where it's going. I have to be very careful.

Indirect branches. so I keep addresses somewhere in that branch, so that I don't

know where it's going. Something computed go to [UNINTELLIGIBLE].

Large switch statement. It's just spaghetti code. We have no idea where it would end up and compile at us, and we can't get anywhere in this switch statement. Either [UNINTELLIGIBLE] you might know some order of going through that, it doesn't work.

If you are looped with [UNINTELLIGIBLE] breaks and very complex things in the compiler, sometimes it'll give up. When the loop bounds are unknown, you'd assume it could be anything. Whereas when loop bounds are known, as you saw in the first set of examples, you can take advantages a lot more, and you can do a lot more aggressive things, or not care about cases because I know that. But in this unknown loop bounds, you have to be a lot more careful of that.

And conditions where branch is not analyzable. So if you have branch condition, if you don't know what's happening in the branch, I might not be able to take advantages or think how to do the branch well. So those are the things that I have to worry about.

The other thing is data accessors, so that means who else can read and write the data. So I am touching the data item, and I need to know that, between the two points I am looking at the data, nobody else go and muck with my data, or use my data. Because when I look at the data, [UNINTELLIGIBLE] something, I want to make sure that's the only way that data can be accessed because, as you know, most of the things are in memory.

So normally compiler [UNINTELLIGIBLE] is called def-use chains, so defined to use, so we say that thing that defined here is going to get used here, and nothing comes in between that. And that information is that's how the compiler [UNINTELLIGIBLE]. That's something we call dependence vectors. We might talk a little bit about that when you go into parallel execution.

So what makes it very hard for compiler to analyze this? For example, address taken variables, so if you write and hack with C, you can say, OK, there's a variable.

There's a variable here, I'm taking the address of that.

Suddenly, that means somebody else has the address to the variable. That means anybody else can suddenly jump in and overwrite you, and there's a lot of possibilities of doing that. And suddenly compiler says wait a minute, that variable, even though I assigned the variable here, I'm using it here, in between. Somebody else might touch it even though it might not use the same name because somebody has that address to that.

OK, so that's a hard thing. Global variables, sometimes, because between function, I don't know. Some other function might go and change it.

Parameters are really hard. Like for example, remember when we had a program, and we had something like copying same array to the same, even though parameters say X and Y. I might send the same or overlapping regions into two different parameters even though it looks like two different names. They're not two different things. They're actually overlapping at some point.

And so you had to assume, even if you have two different parameters point into memory, they might be the same thing. And that's the worst case, even though a lot of times, nobody does that. Nobody gives the same things multiple names, but it's possible. If it is possible, compilers deal with it.

Either it has to generate code to test all these cases, is it overlapping, if not, do something. If it is overlapping, do something slower, like the code we showed when you are vectorizing. You treat it like this huge number of different cases, but unless you do something like that, you can't optimize, and complex programs, it's very hard to do that.

A lot of times, pointers create issues in here because the problem with pointers is what you call it point aliasing, because pointers, you can add any value to a pointer and you have no idea if you had a very large value. It can be anywhere in memory because if you have a pointer, you have a point in the memory you can add anything, subtract anything. The world is yours, and C gives you this ability to go all

over the world and kind of mapping the world, and some programs do that.

And so the compiler says, oh, it's a point. I don't know where it is. I just have to leave it alone because some guy, probably 0.001% of the world programmers will do something crazy, and everybody has to pay the price. So this is what makes programming hard.

And the final thing is there's a thing called [UNINTELLIGIBLE] types. When you go to parallel programming you realize, because normally compilers keep normal values are in the memory. Compiler can [UNINTELLIGIBLE] the value into register and keep operating in the register, and at some point, put it back to memory. But if you're running a parallel program, somebody else might want to look at that value, and if it isn't registered, you don't have that value in the right place. It's somewhere else, so you get a stale copy because you have moved it.

What I'm trying to say is, look, you have to always keep it in memory. You can't take it out. You can't just modify it, but you can move it somewhere else the faster place to do things to it because somebody else might be looking at it. And so what that means is compilers give up it's hands and say, look, I can't do anything.

So we are a little bit early. I have yet another huge session in here at-- OK, we have to go through this thing. Good.

I think now we are going to go about and see how you guys did in the class exam. OK. And I'm seeing it for the first time, and it looks really nice. Where do you plug this in? Where do you plug this in?

OK, so here is the distribution in there. This was not an easy exam, and in fact, we compared how you guys did last year, and you guys have done a lot better than I think the first exam in last year. So basically, we have a median about 70, somewhere here, and a nice tight grouping in here, which is really good. And so what we have is, we have exams back. Take a look. And I think--

GUEST SPEAKER: I'd like to make one comment about [INAUDIBLE].

PROFESSOR: OK, sure.

GUEST SPEAKER: [INAUDIBLE]. So not surprisingly, I graded the problem on the cache oblivious algorithm doing the recursion tree. There is a common mistake that many people made, which I wanted to explain why it's wrong because so many people made this mistake. They got it almost all right, and then they made this mistake.

So it's basically an understanding of recurrence. So the recurrences I recall was q of r is equal to square root of r over b if square root of r is less than CM for C, et cetera. OK? And then otherwise, it was 2q of r over 2 plus theta 1.

Now, what people did in their recursion tree-- first of all, some people didn't recognize that what goes in the recursion tree is this value, the number of cache misses. So the recursion tree is going to look like theta 1, or you can leave out the thetas if you want to put them in at the end. Theta 1, theta 1, et cetera. So many people got this, and then the question is what happens when it hits the leaf. OK? So when it hits a leaf, many people correctly got that you can't mess around with constants.

You have to be very careful of constants if they're in an exponent, that you hit the leaf when square root of r becomes less than c over m, in which case the cost is going to be square root of r over b. So what they did was the incorrect thing, was they put square root of r over b here. Why is that wrong?

[INTERPOSING VOICES]

GUEST SPEAKER: It's the wrong r. Right? OK, it's the wrong r. This r is the r here on the right-hand side. It's not the one here.

It's the r if r is sufficiently small, that's the value you're taking. But we're expanding an r from the top here. So what's the value that should go here? OK, cm is the value that should go here. OK? The value that should go here is cm. Yes?

AUDIENCE: [INAUDIBLE] two r's can actually follow the right-side? And then it's very close to where they're written the same but are spoken differently.

GUEST SPEAKER: Well, when you say-- what do you mean?

AUDIENCE: [INAUDIBLE].

GUEST SPEAKER: There's an r here.

AUDIENCE: And it's different from the other r--

GUEST SPEAKER: No, it's the same r. The question is there, r is a variable. So it'd be nice if the r were constant, but it's not. It's a variable. And so the point is the point where you plug it in here, you've got to plug in, not the variable, you've got to plug in the value.

AUDIENCE: You just said for r [UNINTELLIGIBLE].

GUEST SPEAKER: It's a variable. You have to plug in the value of the variable at this point if you're going to solve the recurrence. Putting an r here, we're trying to I say, this whole thing is q of r. And we started out, if we did the development of the tree, which is the safest thing to do, you get theta 1 plus q of r over 2, and you keep going down until your value for r satisfies this condition. At that point, what's the value for r? OK?

You can't then say it's the same r that you started with. It's not this r, and that's because r is a variable, not because of anything else. r is a variable, and we're using the r. This is a question of understanding of the recurrence. So in any case, that was a common mistake that people make.

The other minor error that people made on that problem, that most people made, was in describing where do you get this recurrence, they left out the fact is why is it going to be square root of r over b. It's really because na is approximately nb because the way that the code works, we're keeping na and nb to within a factor of two of each other. OK? And so if you didn't mention that, you lost a point. It wasn't a big deal, but many people didn't neglect that very important statement.

Overall, people did very well on this problem. Overall, you'll see people got a lot of partial credit on it.