CHARLES E. LEISERSON: Hi, it's my great pleasure to introduce, again, TB Schardl. TB is not only a fabulous, world-class performance engineer, he is a world-class performance meta engineer. In other words, building the tools and such to make it so that people can engineer fast code. And he's the author of the technology that we're using in our compiler, the taper technology that's in the open compiler for parallelism.

So he implemented all of that, and all the optimizations, and so forth, which has greatly improved the quality of the programming environment. So today, he's going to talk about something near and dear to his heart, which is compilers, and what they can and cannot do.

TAO B. SCHARDL: Great, thank you very much for that introduction. Can everyone hear me in the back? Yes, great. All right, so as I understand it, last lecture you talked about multi-threaded algorithms. And you spent the lecture studying those algorithms, analyzing them in a theoretical sense, essentially analyzing their asymptotic running times, work and span complexity.

This lecture is not that at all. We're not going to do that kind of math anywhere in the course of this lecture. Instead, this lecture is going to take a look at compilers, as professor mentioned, and what compilers can and cannot do.

So the last time, you saw me standing up here was back in lecture five. And during that lecture we talked about LLVM IR and x8664 assembly, and how C code got translated into assembly code via LLVM IR. In this lecture, we're going to talk more about what happens between the LLVM IR and assembly stages. And, essentially, that's what happens when the compiler is allowed to edit and optimize the code in its IR representation, while it's producing the assembly.

So last time, we were talking about this IR, and the assembly. And this time, they called the compiler guy back, I suppose, to tell you about the boxes in the middle. Now, even though you're predominately dealing with C code within this class, I hope that some of the lessons from today's lecture you will be able to take away into any job that you pursue in the future,

because there are a lot of languages today that do end up being compiled, C and C++, Rust, Swift, even Haskell, Julia, Halide, the list goes on and on.

And those languages all get compiled for a variety of different what we call backends, different machine architectures, not just x86-64. And, in fact, a lot of those languages get compiled using very similar compilation technology to what you have in the Clang LLVM compiler that you're using in this class. In fact, many of those languages today are optimized by LLVM itself.

LLVM is the internal engine within the compiler that actually does all of the optimization. So that's my hope, that the lessons you'll learn here today don't just apply to 172. They'll, in fact, apply to software that you use and develop for many years on the road.

But let's take a step back, and ask ourselves, why bother studying the compiler optimizations at all? Why should we take a look at what's going on within this, up to this point, black box of software? Any ideas? Any suggestions? In the back?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: You can avoid manually trying to optimize things that the compiler will do for you, great answer. Great, great answer. Any other answers?

AUDIENCE: You learn how to best write your code to take advantages of the compiler optimizations.

TAO B. SCHARDL: You can learn how to write your code to take advantage of the compiler optimizations, how to suggest to the compiler what it should or should not do as you're constructing your program, great answer as well. Very good, in the front.

AUDIENCE: It might help for debugging if the compiler has bugs.

TAO B. SCHARDL: It can absolutely help for debugging when the compiler itself has bugs. The compiler is a big piece of software. And you may have noticed that a lot of software contains bugs. The compiler is no exception.

And it helps to understand where the compiler might have made a mistake, or where the compiler simply just didn't do what you thought it should be able to do. Understanding more of what happens in the compiler can demystify some of those oddities. Good answer. Any other thoughts?

AUDIENCE: It's fun.

TAO B. SCHARDL: It's fun. Well, OK, so in my completely biased opinion, I would agree that it's fun to understand what the compiler does. You may have different opinions. That's OK. I won't judge.

So I put together a list of reasons why, in general, we may care about what goes on inside the compiler. I highlighted that last point from this list, my bad. Compilers can have a really big impact on software.

It's kind of like this. Imagine that you're working on some software project. And you have a teammate on your team he's pretty quiet but extremely smart. And what that teammate does is whenever that teammate gets access to some code, they jump in and immediately start trying to make that code work faster.

And that's really cool, because that teammate does good work. And, oftentimes, you see that what the teammate produces is, indeed, much faster code than what you wrote. Now, in other industries, you might just sit back and say, this teammate does fantastic work.

Maybe they don't talk very often. But that's OK. Teammate, you do you.

But in this class, we're performance engineers. We want to understand what that teammate did to the software. How did that teammate get so much performance out of the code? The compiler is kind of like that teammate. And so understanding what the compiler does is valuable in that sense.

As mentioned before, compilers can save you performance engineering work. If you understand that the compiler can do some optimization for you, then you don't have to do it yourself. And that means that you can continue writing simple, and readable, and maintainable code without sacrificing performance.

You can also understand the differences between the source code and whatever you might see show up in either the LLVM IR or the assembly, if you have to look at the assembly language produced for your executable. And compilers can make mistakes.

Sometimes, that's because of a genuine bug in the compiler. And other times, it's because the compiler just couldn't understand something about what was going on. And having some insight into how the compiler reasons about code can help you understand why those mistakes were made, or figure out ways to work around those mistakes, or let you write

meaningful bug reports to the compiler developers.

And, of course, understanding computers can help you use them more effectively. Plus, I think it's fun. So the first thing to understand about a compiler is a basic anatomy of how the compiler works. The compiler takes as input LLVM IR.

And up until this point, we thought of it as just a big black box. That does stuff to the IR, and out pops more LLVM IR, but it's somehow optimized. In fact, what's going on within that black box the compiler is executing a sequence of what we call transformation passes on the code.

Each transformation pass takes a look at its input, and analyzes that code, and then tries to edit the code in an effort to optimize the code's performance. Now, a transformation pass might end up running multiple times. And those passes run in some order.

That order ends up being a predetermined order that the compiler writers found to work pretty well on their tests. That's about the level of insight that went into picking the order. It seems to work well.

Now, some good news, in terms of trying to understand what the compiler does, you can actually just ask the compiler, what did you do? And you've already used this functionality, as I understand, in some of your assignments. You've already asked the compiler to give you a report specifically about whether or not it could vectorize some code.

But, in fact, LLVM, the compiler you have access to, can produce reports not just for factorization, but for a lot of the different transformation passes that it tries to perform. And there's some syntax that you have to pass to the compiler, some compiler flags that you have to specify in order to get those reports. Those are described on the slide.

I won't walk you through that text. You can look at the slides afterwards. At the end of the day, the string that you're passing is actually a regular expression. If you know what regular expressions are, great, then you can use that to narrow down the search for your report. If you don't, and you just want to see the whole report, just provide dot star as a string and you're good to go.

That's the good news. You can get the compiler to tell you exactly what it did. The bad news is that when you ask the compiler what it did, it will give you a report.

And the report looks something like this. In fact, I've highlighted most of the report for this

particular piece of code, because the report ends up being very long. And as you might have noticed just from reading some of the texts, there are definitely English words in this text. And there are pointers to pieces of code that you've compiled.

But it is very jargon, and hard to understand. This isn't the easiest report to make sense of. OK, so that's some good news and some bad news about these compiler reports.

The good news is, you can ask the compiler. And it'll happily tell you all about the things that it did. It can tell you about which transformation passes were successfully able to transform the code.

It can tell you conclusions that it drew about its analysis of the code. But the bad news is, these reports are kind of complicated. They can be long. They use a lot of internal compiler jargon, which if you're not familiar with that jargon, it makes it hard to understand.

It also turns out that not all of the transformation passes in the compiler give you these nice reports. So you don't get to see the whole picture. And, in general, the reports don't really tell you the whole story about what the compiler did or did not do. And we'll see another example of that later on.

So part of the goal of today's lecture is to get some context for understanding the reports that you might see if you pass those flags to the compiler. And the structure of today's lecture is basically divided up into two parts.

First, I want to give you some examples of compiler optimizations, just simple examples so you get a sense as to how a compiler mechanically reasons about the code it's given, and tries to optimize that code. We'll take a look at how the compiler optimizes a single scalar value, how it can optimize a structure, how it can optimize function calls, and how it can optimize loops, just simple examples to give some flavor.

And then the second half of lecture, I have a few case studies for you which get into diagnosing ways in which the compiler failed, not due to bugs, per se, but simply didn't do an optimization you might have expected it to do. But, to be frank, I think all those case studies are really cool. But it's not totally crucial that we get through every single case study during today's lecture.

The slides will be available afterwards. So when we get to that part, we'll just see how many case studies we can cover. Sound good? Any questions so far?

All right, let's get to it. Let's start with a quick overview of compiler optimizations. So here is a summary of the various-- oh, I forgot that I just copied this slide from a previous lecture given in this class. You might recognize this slide I think from lecture two. Sorry about that.

That's OK. We can fix this. We'll just go ahead and add this slide right now. We need to change the title. So let's cross that out and put in our new title.

OK, so, great, and now we should double check these lists and make sure that they're accurate. Data structures, we'll come back to data structures. Loops, hoisting, yeah, the compiler can do hoisting.

Sentinels, not really, the compiler is not good at sentinels. Loop unrolling, yeah, it absolutely does loop unrolling. Loop fusion, yeah, it can, but there are some restrictions that apply. Your mileage might vary.

Eliminate waste iterations, some restrictions might apply. OK, logic, constant folding and propagation, yeah, it's good on that. Common subexpression elimination, yeah, I can find common subexpressions, you're fin there.

It knows algebra, yeah good. Short circuiting, yes, absolutely. Ordering tests, depends on the tests-- I'll give it to the compiler. But I'll say, restrictions apply.

Creating a fast path, compilers aren't that smart about fast paths. They come up with really boring fast paths. I'm going to take that off the list. Combining tests, again, it kind of depends on the tests.

Functions, compilers are pretty good at functions. So inling, it can do that. Tail recursion elimination, yes, absolutely. Coarsening, not so much. OK, great.

Let's come back to data structures, which we skipped before. Packing, augmentation-- OK, honestly, the compiler does a lot with data structures but really none of those things. The compiler isn't smart about data structures in that particular way.

Really, the way that the compiler is smart about data structures is shown here, if we expand this list to include even more compiler optimizations. Bottom line with data structures, the compiler knows a lot about architecture. And it really has put a lot of effort into figuring out how to use registers really effectively.

Reading and writing and register is super fast. Touching memory is not so fast. And so the compiler works really hard to allocate registers, put anything that lives in memory ordinarily into registers, manipulate aggregate types to use registers, as we'll see in a couple of slides, align data that has to live in memory. Compilers are good at that.

Compilers are also good at loops. We already saw some example optimization on the previous slide. It can vectorize. It does a lot of other cool stuff.

Unswitching is a cool optimization that I won't cover here. Idiom replacement, it finds common patterns, and does something smart with those. Vision, skewing, tiling, interchange, those all try to process the iterations of the loop in some clever way to make stuff go fast. And some restrictions apply. Those are really in development in LLVM.

Logic, it does a lot more with logic than what we saw before. It can eliminate instructions that aren't necessary. It can do strength reduction, and other cool optimization. I think we saw that one in the Bentley slides.

It gets rid of dead code. It can do more idiom replacement. Branch reordering is kind like reordering tests. Global value numbering, another cool optimization that we won't talk about today.

Functions, it can do more on switching. It can eliminate arguments that aren't necessary. So the compiler can do a lot of stuff for you.

And at the end the day, writing down this whole list is kind of a futile activity because it changes over time. Compilers are a moving target. Compiler developers, they're software engineers like you and me.

And they're clever. And they're trying to apply all their clever software engineering practice to this compiler code base to make it do more stuff. And so they are constantly adding new optimizations to the compiler, new clever analyses, all the time.

So, really, what we're going to look at today is just a couple examples to get a flavor for what the compiler does internally. Now, if you want to follow along with how the compiler works, the good news is, by and large, you can take a look at the LLVM IR to see what happens as the compiler processes your code. You don't need to look out the assembly.

That's generally true. But there are some exceptions. So, for example, if we have these three

snippets of C code on the left, and we look at what your LLVM compiler generates, in terms of the IR, we can see that there are some optimizations reflected, but not too many interesting ones.

The multiply by 8 turns into a shift left operation by 3, because 8 is a power of 2. That's straightforward. Good, we can see that in the IR.

The multiply by 15 still looks like a multiply by 15. No changes there. The divide by 71 looks like a divide by 71. Again, no changes there.

Now, with arithmetic ops, the difference between what you see in the LLVM IR and what you see in the assembly, this is where it's most pronounced, at least in my experience, because if we take a look at these same snippets of C code, and we look at the corresponding x86 assembly for it, we get the stuff on the right. And this looks different.

Let's pick through what this assembly code does one line at a time. So the first one in the C code, takes the argument n, and multiplies it by 8. And then the assembly, we have this LEA instruction. Anyone remember what the LEA instruction does?

I see one person shaking their head. That's a perfectly reasonable response. Yeah, go for it? Load effective address, what does that mean? Load the address, but don't actually access memory.

Another way to phrase that, do this address calculation. And give me the result of the address calculation. Don't read or write memory at that address. Just do the calculation. That's what loading an effective address means, essentially.

But you're exactly right. The LEA instruction does an address calculation, and stores the result in the register on the right. Anyone remember enough about x86 address calculations to tell me how that LEA in particular works, the first LEA on the slide? Yeah?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: But before the first comma, in this case nothing, gets added to the product of the second two arguments in those parens. You're exactly right. So this LEA takes the value 8, multiplies it by whatever is in register RDI, which holds the value n. And it stores the result into AX.

So, perfect, it does a multiply by 8. The address calculator is only capable of a small range of

operations. It can do additions. And it can multiply by 1, 2, 4, or 8. That's it.

So it's a really simple circuit in the hardware. But it's fast. It's optimized heavily by modern processors. And so if the compiler can use it, they tend to try to use these LEA instructions. So good job.

How about the next one? Multiply by 15 turns into these two LEA instructions. Can anyone tell me how these work?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: You're basically multiplying by 5 and multiplying by 3, exactly right. We can step through this as well. If we look at the first LEA instruction, we take RDI, which stores the value n. We multiply that by 4.

We add it to the original value of RDI. And so that computes 4 times n, plus n, which is five times n. And that result gets stored into AX. Could, we've effectively multiplied by 5.

The next instruction takes whatever is in REX, which is now 5n, multiplies that by 2, adds it to whatever is currently in REX, which is once again 5n. So that computes 2 times 5n, plus 5n, which is 3 times 5n, which is 15n. So just like that, we've done our multiply with two LEA instructions.

How about the last one? In this last piece of code, we take the arguments in RDI. We move it into EX. We then move the value 3,871,519,817, and put that into ECX, as you do. We multiply those two values together.

And then we shift the product right by 38. So, obviously, this divides by 71. Any guesses as to how this performs the division operation we want? Both of you answered.

I might still call on you. give a little more time for someone else to raise their hand. Go for it.

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: It has a lot to do with 2 to the 38, very good. Yeah, all right, any further guesses before I give the answer away? Yeah, in the back?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: Kind of. So this is what's technically called a magic number. And, yes, it's technically called a magic number. And this magic number is equal to 2 to the 38, divided by 71, plus 1 to deal with some rounding effects.

What this code does is it says, let's compute n divided by 71, by first computing n divided by 71, times 2 to the 38, and then shifting off the lower 38 bits with that shift right operation. And by converting the operation into this, it's able to replace the division operation with a multiply. And if you remember, hopefully, from the architecture lecture, multiply operations, they're not the cheapest things in the world.

But they're not too bad. Division is really expensive. If you want fast code, never divide. Also, never compute modulus, or access memory. Yeah, question?

AUDIENCE: Why did you choose 38?

TAO B. SCHARDL: Why did I choose 38? I think it shows 38 because 38 works. There's actually a formula for-- pretty much it doesn't want to choose a value that's too large, or else it'll overflow. And it doesn't want to choose a value that's too small, or else you lose precision.

So it's able to find a balancing point. If you want to know more about magic numbers, I recommend checking out this book called *Hackers Delight.* For any of you who are familiar with this book, it is a book full of bit tricks. Seriously, that's the entire book. It's just a book full of bit tricks.

And there's a whole section in there describing how you do division by various constants using multiplication, either signed or unsigned. It's very cool. But magic number to convert a division into a multiply, that's the kind of thing that you might see from the assembly.

That's one of these examples of arithmetic operations that are really optimized at the very last step. But for the rest of the optimizations, fortunately we can focus on the IR. Any questions about that so far?

Cool. OK, so for the next part of the lecture, I want to show you a couple example optimizations in terms of the LLVM IR. And to show you these optimizations, we'll have a little bit of code that we'll work through, a running example, if you will.

And this running example will be some code that I stole from I think it was a serial program that simulates the behavior of n massive bodies in 2D space under the law of gravitation. So

we've got a whole bunch of point masses. Those point masses have varying masses.

And we just want to simulate what happens due to gravity as these masses interact in the plane. At a high level, the n body code is pretty simple. We have a top level simulate routine, which just loops over all the time steps, during which we want to perform this simulation.

And at each time step, it calculates the various forces acting on those different bodies. And then it updates the position of each body, based on those forces. In order to do that calculation. It has some internal data structures, one to represent each body, which contains a couple of vector types.

And we define our own vector type to store to double precision floating point values. Now, we don't need to see the entire code in order to look at some compiler optimizations. The one routine that we will take a look at is this one to update the positions. This is a simple loop that takes each body, one at a time, computes the new velocity on that body, based on the forces acting on that body, and uses vector operations to do that.

Then it updates the position of that body, again using these vector operations that we've defined. And then it stores the results into the data structure for that body. So all these methods with this code make use of these basic routines on 2D vectors, points in x, y, or points in 2D space.

And these routines are pretty simple. There is one to add two vectors. There's another to scale a vector by a scalar value. And there's a third to compute the length, which we won't actually look at too much.

Everyone good so far? OK, so let's try to start simple. Let's take a look at just one of these one line vector operations, vec scale. All vec scale does is it takes one of these vector inputs at a scalar value a.

And it multiplies x by a, and y by a, and stores the results into a vector type, and return to it. Great, couldn't be simpler. If we compile this with no optimizations whatsoever, and we take a look at the LLVM IR, we get that, which is a little more complicated than you might imagine.

The good news, though, is that if you turn on optimizations, and you just turn on the first level of optimization, just 01, whereas we got this code before, now we get this, which is far, far simpler, and so simple I can blow up the font size so you can actually read the code on the

slide. So to see, again, no optimizations, optimizations.

So a lot of stuff happened to optimize this simple function. We're going to see what those optimizations actually were. But, first, let's pick apart what's going on in this function.

We have our vec scale routine in LLVM IR. It takes a structure as its first argument. And that's represented using two doubles. It takes a scalar as the second argument.

And what the operation does is it multiplies those two fields by the third argument, the double A. It then packs those values into a struct that'll return. And, finally, it returns that struct. So that's what the optimized code does.

Let's see actually how we get to this optimized code. And we'll do this one step at a time. Let's start by optimizing the operations on a single scalar value. That's why I picked this example.

So we go back to the 00 code. And we just pick out the operations that dealt with that scalar value. We our scope down to just these lines. So the argument double A is the final argument in the list.

And what we see is that within the vector scale routine, compiler to 0, we allocate some local storage. We store that double A into the local storage. And then later on, we'll load the value out of the local storage before the multiply. And then we load it again before the other multiply.

OK, any ideas how we could make this code faster? Don't store in memory, what a great idea. How do we get around not storing it in memory?

Saving a register. In particular, what property of LLVM IR makes that really easy? There are infinite registers. And, in fact, the argument is already in a register.

It's already in the register percent two, if I recall. So we don't need to move it into a register. It's already there. So how do we go about optimizing that code in this case?

Well, let's find the places where we're using the value. And we're using the value loaded from memory. And what we're going to do is just replace those loads from memory with the original argument. We know exactly what operation we're trying to do. We know we're trying to do a multiply by the original parameter.

So we just find those two uses. We cross them out. And we put in the input parameter in its place. That make sense? Questions so far? Cool.

So now, those multipliers aren't using the values returned by the loads. How further can we optimize this code? Delete the loads. What else can we delete?

So there's no address calculation here just because the code is so simple, but good insight. The allocation and the store, great. So those loads are dead code. The store is dead code. The allocation is dead code.

We eliminate all that dead code. We got rid of those loads. We just used the value living in the register.

And we've already eliminated a bunch of instructions. So the net effect of that was to turn the code optimizer at 00 that we had in the background into the code we have in the foreground, which is slightly shorter, but not that much. So it's a little bit faster, but not that much faster.

How do we optimize this function further? Do it for every variable we have. In particular, the only other variable we have is a structure that we're passing in. So we want to do this kind of optimization on the structure. Make sense?

So let's see how we optimize this structure. Now, the problem is that structures are harder to handle than individual scalar values, because, in general, you can't store the whole structure in just a single register. It's more complicated to juggle all the data within a structure.

But, nevertheless, let's take a look at the code that operates on the structure, or at least operates on the structure that we pass in to the function. So when we eliminate all the other code, we see that we've got an allocation. See if I animations here, yeah, I do.

We have an allocation. So we can store the structure onto the stack. Then we have an address calculation that lets us store the first part of the structure onto the stack.

We have a second address calculation to store the second field on the stack. And later on, when we need those values, we load the first field out of memory. And we load the second field out of memory. It's a very similar pattern to what we had before, except we've got more going on in this case.

So how do we go about optimizing this structure? Any ideas, high level ideas? Ultimately, we want to get rid of all of the memory references and all that storage for the structure. How do we reason through eliminating all that stuff in a mechanical fashion, based on what we've seen

so far? Go for it.

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: They are passed in using separate parameters, separate registers if you will, as a quirk of how LLVM does it. So given that insight, how would you optimize it?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: Cross out percent 12, percent 6, and put in the relevant field. Cool. Let me phrase that a little bit differently. Let's do this one field at a time.

We've got a structure, which has multiple fields. Let's just take it one step at a time. All right, so let's look at the first field. And let's look at the operations that deal with the first field.

We have, in our code, in our LLVM IR, some address calculations that refer to the same field of the structure. In this case, I believe it's the first field, yes.

And, ultimately, we end up loading from this location in local memory. So what value is this load going to retrieve? How do we know that both address calculations refer to the same field?

Good question. What we do in this case is very careful analysis of the math that's going on. We know that the alga, the location in local memory, that's just a fixed location. And from that, we can interpret what each of the instructions does in terms of an address calculation. And we can determine that they're the same value.

So we have this location in memory that we operate on. And before you do a multiply, we end up loading from that location in memory. So what value do we know is going to be loaded by that load instruction? Go for it.

AUDIENCE: So what we're doing right now is taking some value, and then storing it, and then getting it back out, and putting it back.

TAO B. SCHARDL: Not putting it back, but we don't you worry about putting it back.

AUDIENCE: So we don't need to put it somewhere just to take it back out?

TAO B. SCHARDL: Correct. Correct. So what are we multiplying in that multiply, which value? First element of the struct. It's percent zero. It's the value that we stored right there.

That makes sense? Everyone see that? Any questions about that?

All right, so we're storing the first element of the struct into this location. Later, we load it out of that same location. Nothing else happened to that location.

So let's go ahead and optimize it just the same way we optimize the scalar. We see that we use the result of the load right there. But we know that load is going to return the first field of our struct input.

So we'll just cross it out, and replace it with that field. So now we're not using the result of that load. What do we get to do as the compiler? I can tell you know the answer.

Delete the dead code, delete all of it. Remove the now dead code, which is all those address calculations, as well as the load operation, and the store operation. And that's pretty much it. Yeah, good.

So we replace that operation. And we got rid of a bunch of other code from our function. We've now optimized one of the two fields in our struct. What do we do next?

Optimize the next one. That happened similarly. I won't walk you through that a second time. We find where we're using the result of that load.

We can cross it out, and replace it with the appropriate input, and then delete all the relevant dead code. And now, we get to delete the original allocation because nothing's getting stored to that memory. That make sense? Any questions about that? Yeah?

AUDIENCE: So when we first compile it to LLVM IR, does it unpack the struct and just put in separate parameters?

TAO B. SCHARDL: When we first compiled LLVM IR, do we unpack the struct and pass in the separate parameters?

AUDIENCE: Like, how we have three parameters here that are doubled. Wasn't our original C code just a struct of vectors in the double?

TAO B. SCHARDL: So LLVM IR in this case, when we compiled it as zero, decided to pass it as separate parameters, just as it's representation. So in that sense, yes. But it was still doing the standard, create some local storage, store the parameters on to local storage, and then all

operations just read out of local storage.

It's the standard thing that the compiler generates when it's asked to compile C code. And with no other optimizations, that's what you get. That makes sense? Yeah?

AUDIENCE: What are all the align eights?

TAO B. SCHARDL: What are all the aligned eights doing? The align eights are attributes that specify the alignment of that location in memory. This is alignment information that the compiler either determines by analysis, or implements as part of a standard.

So they're specifying how values are aligned in memory. That matters a lot more for ultimate code generation, unless we're able to just delete the memory references altogether. Make sense? Cool. Any other questions?

All right, so we optimized the first field. We optimize the second field in a similar way. Turns out, there's additional optimizations that need to happen in order to return a structure from this function.

Those operations can be optimized in a similar way. They're shown here. We're not going to go through exactly how that works. But at the end of the day, after we've optimized all of that code we end up with this.

We end up with our function compiled at 01. And it's far simpler. I think it's far more intuitive. This is what I would imagine the code should look like when I wrote the C code in the first place.

Take your input. Do a couple of multiplications. And then it does them operations to create the return value, and ultimately return that value.

So, in summary, the compiler works hard to transform data structures and scalar values to store as much as it possibly can purely within registers, and avoid using any local storage, if possible. Everyone good with that so far? Cool.

Let's move on to another optimization. Let's talk about function calls. Let's take a look at how the compiler can optimize function calls.

By and large, these optimizations will occur if you pass optimization level 2 or higher, just FYI. So from our original C code, we had some lines that performed a bunch of vector operations.

We had a vec add that added two vectors together, one of which was the result of a vec scale, which scaled the result of a vec add by some scalar value.

So we had this chain of calls in our code. And if we take a look at the code compile that was 0, what we end up with is this snippet shown on the bottom, which performs some operations on these vector structures, does this multiply operation, and then calls this vector scale routine, the vector scale routine that we decide to focus on first. So any ideas for how we go about optimizing this?

So to give you a little bit of a hint, what the compiler sees when it looks at that call is it sees a snippet containing the call instruction. And in our example, it also sees the code for the vec scale function that we were just looking at. And we're going to suppose that it's already optimized vec scale as best as it can. It's produced this code for the vec scale routine.

And so it sees that call instruction. And it sees this code for the function that's being called. So what could the compiler do at this point to try to make the code above even faster?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: You're exactly right. Remove the call, and just put the body of the vec scale code right there in place of the call. It takes a little bit of effort to pull that off. But, roughly speaking, yeah, we're just going to copy and paste this code in our function into the place where we're calling the function.

And so if we do that simple copy paste, we end up with some garbage code as an intermediate. We had to do a little bit of renaming to make everything work out. But at this point, we have the code from our function in the place of that call. And now, we can observe that to restore correctness, we don't want to do the call.

And we don't want to do the return that we just pasted in place. So we'll just go ahead and remove both that call and the return. That is called function inlining.

We identify some function call, or the compiler identifies some function call. And it takes the body of the function, and just pastes it right in place of that call. Sound good? Make sense? Anyone confused? Raise your hand if you're confused.

Now, once you've done some amount of function inlining, we can actually do some more

optimizations. So here, we have the code after we got rid of the unnecessary call and return. And we have a couple multiply operations sitting in place. That looks fine.

But if we expand our scope just a little bit, what we see, so we have some operations happening that were sitting there already after the original call. What the compiler can do is it can take a look at these instructions. And long story short, it realizes that all these instructions do is pack some data into a structure, and then immediately unpack the structure.

So it's like you put a bunch of stuff into a bag, and then immediately dump out the bag. That was kind of a waste of time. That's kind of a waste of code. Let's get rid of it.

Those operations are useless. Let's delete them. The compiler has a great time deleting dead code. It's like it's what it lives to do.

All right, now, in fact, in the original code, we didn't just have one function call. We had a whole sequence of function calls. And if we expand our LLVM IR snippet even a little further, we can include those two function calls, the original call to vec ad, followed by the code that we've now optimized by inlining, ultimately followed by yet another call to vec add.

Minor spoiler, the vec add routine, once it's optimized, looks pretty similar to the vec scalar routine. And, in particular, it has comparable size to the vector scale routine. So what's the compiler is going to do to those to call sites?

Inline it, do more inlining, inlining is great. We'll inline these functions as well, and then remove all of the additional, now-useless instructions. We'll walk through that process. The result of that process looks something like this.

So in the original C code, we had this vec add, which called the vec scale as one of its arguments, which called the vec add is one of its arguments. And what we end up with in the optimized IR is just a bunch of straight line code that performs floating point operations. It's almost as if the compiler took the original C code, and transformed it into the equivalency code shown on the bottom, where it just operates on a whole bunch of doubles, and just does primitive operations.

So function inlining, as well as the additional transformations it was able to perform as a result, together those were able to eliminate all of those function calls. It was able to completely eliminate any costs associated with the function call abstraction, at least in this code. Make sense?

I think that's pretty cool. You write code that has a bunch of function calls, because that's how you've constructed your interfaces. But you're not really paying for those function calls. Function calls aren't the cheapest operation in the world, especially if you think about everything that goes on in terms of the registers and the stack. But the compiler is able to avoid all of that overhead, and just perform the floating point operations we care about.

OK, well, if function inlining is so great, and it enables so many great optimizations, why doesn't the compiler just inline every function call? Go for it. Recursion, it's really hard to inline a recursive call. In general, you can't inline a function into itself, although it turns out there are some exceptions. So, yes, recursion creates problems with function inlining. Any other thoughts? In the back.

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: You're definitely on to something. So we had to do a bunch of this renaming stuff when we inlined the first time, and when we inlined every single time. And even though LLVM IR has an infinite number of registers, the machine doesn't.

And so all of that renaming does create a problem. But there are other problems as well of a similar nature when you start inlining all those functions. For example, you copy pasted a bunch of code. And that made the original call site even bigger, and bigger, and bigger, and bigger.

And programs, we generally don't think about the space they take in memory. But they do take space in memory. And that does have an impact on performance. So great answer, any other thoughts?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: If your function becomes too long, then it may not fit in instruction cache. And that can increase the amount of time it takes just to execute the function. Right, because you're now not getting hash hits, exactly right.

That's one of the problems with this code size blow up from inlining everything. Any other thoughts? Any final thoughts?

So there are three main reasons why the compiler won't inline every function. I think we touched on two of them here. For some function calls, like recursive calls, it's impossible to inline them, because you can't inline a function into itself. But there are exceptions to that, namely recursive tail calls.

If the last thing in a function is a function call, then it turns out you can effectively inline that function call as an optimization. We're not going to talk too much about how that works. But there are corner cases. But, in general, you can't inline a recursive call.

The compiler has another problem. Namely, if the function that you're calling is in a different castle, if it's in a different compilation unit, literally in a different file that's compiled independently, then the compiler can't very well inline that function, because it doesn't know about the function. It doesn't have access to that function's code.

There is a way to get around that problem with modern compiler technology that involves whole program optimization. And I think there's some backup slides that will tell you how to do that with LLVM. But, in general, if it's in a different compilation unit, it can't be inline. And, finally, as touched on, function inlining can increase code size, which can hurt performance.

OK, so some functions are OK to inline. Other functions could create this performance problem, because you've increased code size. So how does the compiler know whether or not inlining any particular function at a call site could hurt performance? Any guesses? Yeah?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: Yeah. So the compiler has some cost model, which gives it some information about, how much will it cost to inline that function? Is the cost model always right?

It is not. So the answer, how does the compiler know, is, really, it doesn't know. It makes a best guess using that cost model, and other heuristics, to determine, when does it make sense to try to inline a function? And because it's making a best guess, sometimes the compiler guesses wrong.

So to wrap up this part, here are just a couple of tips for controlling function inlining in your own programs. If there's a function that you know must always be inlined, no matter what happens, you can mark that function with a special attribute, namely the always inline attribute.

For example, if you have a function that does some complex address calculation, and it should be inlined rather than called, you may want to mark that with an always inline attribute. Similarly, if you have a function that really should never be inlined, it's never cost effective to inline that function, you can mark that function with the no inline attribute. And, finally, if you want to enable more function inlining in the compiler, you can use link time optimization, or LTO, to enable whole program optimization.

Won't go into that during these slides. Let's move on, and talk about loop optimizations. Any questions so far, before continue? Yeah?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: Sorry?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: Does static inline guarantee you the compiler will always inline it? It actually doesn't. The inline keyword will provide a hint to the compiler that it should think about inlining the function. But it doesn't provide any guarantees. If you want a strong guarantee, use the always inline attribute. Good question, though.

All right, loop optimizations-- you've already seen some loop optimizations. You've seen vectorization, for example. It turns out, the compiler does a lot of work to try to optimize loops. So first, why is that? Why would the compiler engineers invest so much effort into optimizing loops? Why loops in particular?

They're extremely common control structure that also has a branch. Both things are true. I think there's a higher level reason, though, or more fundamental reason, if you will. Yeah?

AUDIENCE: Most of the time, the loop takes up the most time.

TAO B. SCHARDL: Most of the time the loop takes up the most time. You got it. Loops account for a lot of the execution time of programs. The way I like to think about this is with a really simple thought experiment.

Let's imagine that you've got a machine with a two gigahertz processor. We've chosen these values to be easier to think about using mental math. Suppose you've got a two gigahertz processor with 16 cores. Each core executes one instruction per cycle.

And suppose you've got a program which contains a trillion instructions and ample parallelism for those 16 cores. But all of those instructions are simple, straight line code. There are no branches. There are no loops. There no complicated operations like IO.

It's just a bunch of really simple straight line code. Each instruction takes a cycle to execute. The processor executes one instruction per cycle. How long does it take to run this code, to execute the entire terabyte binary?

2 to the 40th cycles for 2 to the 40 instructions. But you're using a two gigahertz processor and 16 cores. And you've got ample parallelism in the program to keep them all saturated. So how much time?

AUDIENCE: 32 seconds.

TAO B. SCHARDL: 32 seconds, nice job. This one has mastered power of 2 arithmetic in one's head. It's a good skill to have, especially in core six.

Yeah, so if you have just a bunch of simple, straight line code, and you have a terabyte of it. That's a lot of code. That is a big binary.

And, yet, the program, this processor, this relatively simple processor, can execute the whole thing in just about 30 seconds. Now, in your experience working with software, you might have noticed that there are some programs that take longer than 30 seconds to run. And some of those programs don't have terabyte size binaries.

The reason that those programs take longer to run, by and large, is loops. So loops account for a lot of the execution time in real programs. Now, you've already seen some loop optimizations.

We're just going to take a look at one other loop optimization today, namely code hoisting, also known as loop invariant code motion. To look at that, we're going to take a look at a different snippet of code from the end body simulation. This code calculates the forces going on each of the end bodies. And it does it with a doubly nested loop.

For all the zero to number of bodies, for all body zero number bodies, as long as you're not looking at the same body, call this add force routine, which calculates to-- calculate the force between those two bodies. And add that force to one of the bodies. That's all that's going on in

this code.

If we translate this code into LLVM IR, we end up with, hopefully unsurprisingly, a doubly nested loop. It looks something like this. The body of the code, the body of the innermost loop, has been lighted, just so things can fit on the slide.

But we can see the overall structure. On the outside, we have some outer loop control. This should look familiar from lecture five, hopefully. Inside of that outer loop, we have an inner loop.

And at the top and the bottom of that inner loop, we have the inner loop control. And within that inner loop, we do have one branch, which can skip a bunch of code if you're looking at the same body for i and j. But, otherwise, we have the loop body of the inner most loop, basic structure.

Now, if we just zoom in on the top part of this doubly-nested loop, just the topmost three basic blocks, take a look at more of the code that's going on here, we end up with something that looks like this. And if you remember some of the discussion from lecture five about the loop induction variables, and what that looks like in LLVM IR, what you find is that for the outer loop we have an induction variable at the very top.

It's that weird fee instruction, once again. Inside that outer loop, we have the loop control for the inner loop, which has its own induction variable. Once again, we have another fee node. That's how we can spot it.

And then we have the body of the innermost loop. And this is just the start of. It it's just a couple address calculations. But can anyone tell me some interesting property about just a couple of these address calculations that could lead to an optimization?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: The first two address calculations only depend on the outermost loop variable, the iteration variable for the outer loop, exactly right. So what can we do with those instructions? Bring them out of the inner loop. Why should we keep computing these addresses in the innermost loop when we could just compute them once in the outer loop?

That optimization is called code hoisting, or loop invariant code motion. Those instructions are invariant to the code in the innermost loop. So you hoist them out.

And once you hoist them out, you end up with a transformed loop that looks something like this. What we have is the same outer loop control at the very top. But now, we're doing some address calculations there. And we no longer have those address calculations on the inside.

And as a result, those hoisted calculations are performed just once per iteration of the outer loop, rather than once per iteration of the inner loop. And so those instructions are run far fewer times. You get to save a lot of running time.

So the effect of this optimization in terms of C code, because it can be a little tedious to look at LLVM IR, is essentially like this. We took this doubly-nested loop in C. We're calling add force of blah, blah, blah, calculate force, blah, blah, blah.

And now, we just move the address calculation to get the ith body that we care about. We move that to the outer. Now, this was an example of loop invariant code motion on just a couple address calculations. In general, the compiler will try to prove that some calculation is invariant across all the iterations of a loop.

And whenever it can prove that, it will try to hoist that code out of the loop. If it can get code out of the body of a loop, that reduces the running time of the loop, saves a lot of execution time. Huge bang for the buck. Make sense? Any questions about that so far?

All right, so just to summarize this part, what can the compiler do? The compiler optimizes code by performing a sequence of transformation passes. All those passes are pretty mechanical.

The compiler goes through the code. It tries to find some property, like this address calculation is the same as that address calculation. And so this load will return the same value as that store, and so on, and so forth. And based on that analysis, it tries to get rid of some dead code, and replace certain register values with other register values, replace things that live in memory with things that just live in registers.

A lot of the transformations resemble Bentley-rule work optimizations that you've seen in lecture two. So as you're studying for your upcoming quiz, you can kind of get two for one by looking at those Bentley-rule optimizations. And one transformation pass, in particular function inlining, was a good example of this.

One transformation can enable other transformations. And those together can compound to

give you fast code. In general, compilers perform a lot more transformations than just the ones we saw today. But there are things that the compiler can't do. Here's one very simple example.

In this case, we're taking another look at this calculate forces routine. Although the compiler can optimize the code by moving address calculations out of loop, one thing that I can't do is exploit symmetry in the problem. So in this problem, what's going on is we're computing the forces on any pair of bodies using the law of gravitation.

And it turns out that the force acting on one body by another is exactly the opposite the force acting on the other body by the one. So F of 12 is equal to minus F of 21. The compiler will not figure that out. The compiler knows algebra.

It doesn't know physics. So it won't be able to figure out that there's symmetry in this problem, and it can avoid wasted operations. Make sense?

All right, so that was an overview of some simple compiler optimizations. We now have some examples of some case studies to see where the compiler can get tripped up. And it doesn't matter if we get through all of these or not. You'll have access to the slides afterwards.

But I think these are kind of cool. So shall we take a look? Simple question-- does the compiler vectorize this loop?

So just to go over what this loop does, it's a simple loop. The function takes two vectors as inputs, or two arrays as inputs, I should say-- an array called y, of like then, and an array x of like then, and some scalar value a. And all that this function does is it loops over each element of the vector, multiplies x of i by the input scalar, adds the product into y's of i. So does the loop vectorize? Yes?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: y and x could overlap. And there is no information about whether they overlap. So do they vectorize? We have a vote for no. Anyone think that it does vectorize?

You made a very convincing argument. So everyone believes that this loop does not vectorize. Is that true? Anyone uncertain?

Anyone unwilling to commit to yes or no right here? All right, a bunch of people are unwilling to commit to yes or no. All right, let's resolve this question.

Let's first ask for the report. Let's look at the vectorization report. We compile it. We pass the flags to get the vectorization report.

And the vectorization report says, yes, it does vectorize this loop, which is interesting, because we have this great argument that says, but you don't know how these addresses fit in memory. You don't know if x and y overlap with each other. How can you possibly vectorize?

Kind of a mystery. Well, if we take a look at the actual compiled code when we optimize this at 02, turns out you can pass certain flags to the compiler, and get it to print out not just the LLVM IR, but the LLVM IR formatted as a control flow graph. And the control flow graph for this simple two line function is the thing on the right, which you obviously can't, read because it's a little bit small, in terms of its text. And it seems have a lot going on.

So I took the liberty of redrawing that control flow graph with none of the code inside, just get a picture of what the structure looks like for this compiled function. And, structurally speaking, it looks like this. And with a bit of practice staring at control flow graphs, which you might get if you spend way too much time working on compilers, you might look at this control flow graph, and think, this graph looks a little too complicated for the two line function that we gave as input.

So what's going on here? Well, we've got three different loops in this code. And it turns out that one of those loops is full of vector operations.

OK, the other two loops are not full of vector operations. That's unvectorized code. And then there's this basic block right at the top that has a conditional branch at the end of it, branching to either the vectorized loop or the unvectorized loop.

And, yeah, there's a lot of other control flow going on as well. But we can focus on just these components for the time being. So what's that conditional branch doing?

Well, we can zoom in on just this one basic block, and actually show it to be readable on the slide. And the basic block looks like this. So let's just study this LLVM IR code.

In this case, we have got the address y stored in register 0. The address of x is stored in register 2. And register 3 stores the value of n. So one instruction at a time, who can tell me what the first instruction in this code does? Yes?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: Gets the address of y. Is that what you said? So it does use the address of y. It's an address calculation that operates on register 0, which stores the address of y. But it's not just computing the address of y.

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: It's getting me the address of the nth element of y. It's adding in whatever is in register 3, which is the value n, into the address of y. So that computes the address y plus n.

This is testing your memory of pointer arithmetic in C just a little bit but. Don't worry. It won't be too rough. So that's what the first address calculation does. What does the next instruction do?

AUDIENCE: It does x plus n.

TAO B. SCHARDL: That computes x plus, very good. How about the next one?

AUDIENCE: It compares whether x plus n and y plus n are the same.

TAO B. SCHARDL: It compares x plus n, versus y plus n.

AUDIENCE: [INAUDIBLE] compares the 33, which is x plus n, and compares it to y. So if x plus n is bigger than y, there's overlap.

TAO B. SCHARDL: Right, so it does a comparison. We'll take that a little more slowly. It does a comparison of x plus n, versus y in checks. Is x plus n greater than y? Perfect. How about the next instruction? Yeah?

AUDIENCE: It compares y plus n versus x.

TAO B. SCHARDL: It compares y plus n, versus x, is y plus n even greater than x. How would the last instruction before the branch? Yep, go for it?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: [INAUDIBLE] one of the results. So this computes the comparison, is x plus n greater than y, bit-wise? And is y plus n greater than x. Fair enough.

So what does the result of that condition mean? I think we've pretty much already spoiled the answer. Anyone want to hear it one last time? We had this whole setup. Go for it.

AUDIENCE: They overlap.

TAO B. SCHARDL: Checks if they overlap. So let's look at this condition in a couple of different situations. If we have x living in one place in memory, and y living in another place in memory, then no matter how we resolve this condition, if we check is both y plus n greater than x, and x plus n greater than y, the results will be false.

But if we have this situation, where x and y overlap in memory some portion of memory, then it turns out that regardless of whether x or y is first, x plus n will be greater than y. y plus n will be greater than x. And the condition will return true.

In other words, the condition returns true, if and only if these portions of memory pointed by x and y alias. So going back to our original looping code, we have a situation where we have a branch based on whether or not they alias. And in one case, it executes the vectorized loop.

And in another case, it executes a non-vectorized code. So returning to our original question, in particular is a vectorized loop if they don't alias. So returning to our original question, does this code get vectorized?

The answer is yes and no. So if you voted yes, you're actually right. If you voted no, and you were persuaded, you were right. And if you didn't commit to an answer, I can't help you.

But that's interesting. The compiler actually generated multiple versions of this loop, due to uncertainty about memory aliasing. Yeah, question?

AUDIENCE: [INAUDIBLE]

TAO B. SCHARDL: So the question is, could the compiler figure out this condition statically while it's compiling the function? Because we know the function is going to get called from somewhere. The answer is, sometimes it can. A lot of times it can't.

If it's not capable of inlining this function, for example, then it probably doesn't have enough information to tell whether or not these two pointers will alias. For example, you're just building a library with a bunch of vector routines. You don't know the code that's going to call this routine eventually.

Now, in general, memory aliasing, this will be the last point before we wrap up, in general,

memory aliasing can cause a lot of issues when it comes to compiler optimization. It can cause the compiler to act very conservatively. In this example, we have a simple serial base case for a matrix multiply routine.

But we don't know anything about the pointers to the C, A, or B matrices. And when we try to compile this and optimize it, the compiler complains that it can't do loop invariant code motion, because it doesn't know anything about these pointers. It could be that the pointer changes within the innermost loop. So it can't move some calculation out to an outer loop.

Compilers try to deal with this statically using an analysis technique called alias analysis. And they do try very hard to figure out, when are these pointers going to alias? Or when are they guaranteed to not alias?

Now, in general, it turns out that alias analysis isn't just hard. It's undecidable. If only it were hard, maybe we'd have some hope. But compilers, in practice, are faced with this undecidable question.

And they try a variety of tricks to get useful alias analysis results in practice. For example, based on information in the source code, the compiler might annotate instructions with various metadata to track this aliasing information. For example, TBAA is aliasing information based on types.

There's some scoping information for aliasing. There is some information that says it's guaranteed not to alias with this other operation, all kinds of metadata. Now, what can you do as a programmer to avoid these issues of memory aliasing? Always annotate your pointers, kids.

Always annotate your pointers. The restrict keyword you've seen before. It tells the compiler, address calculations based off this pointer won't alias with address calculations based off other pointers. The const keyword provides a little more information. It says, these addresses will only be read from. They won't be written to.

And that can enable a lot more compiler optimizations. Now, that's all the time that we have. There are a couple of other cool case studies in the slides. You're welcome to peruse the slides afterwards. Thanks for listening.